Advanced Algorithms and Data Structures
(WS 2019/20)

Martin Held

FB Computerwissenschaften
Universität Salzburg
A-5020 Salzburg, Austria
held@cs.sbg.ac.at

July 29, 2020

UNIVERSITÄT SALZBURG
Computational Geometry and Applications Lab
Instructor (VO): M. Held.

Email: held@cs.sbg.ac.at.

Base-URL: https://www.cosy.sbg.ac.at/~held.

Office: Universität Salzburg, Computerwissenschaften, Rm. 1.20, Jakob-Haringer Str. 2, 5020 Salzburg-Itzling.

Phone number (office): (0662) 8044-6304.

Phone number (secr.): (0662) 8044-6328.
Instructor (PS): P. Palfrader.

Email: palfrader@cs.sbg.ac.at.

Office: Universität Salzburg, Computerwissenschaften, Rm. 0.28, Jakob-Haringer Str. 2, 5020 Salzburg-Itzling.

Phone number (office): (0662) 8044-6326.

Phone number (secr.): (0662) 8044-6328.
Formalia

**URL of course (VO+PS):** Base-URL/teaching/aads/aads.html.

**Lecture times (VO):** Thursday 8\textsuperscript{10}–10\textsuperscript{55}.

**Venue (VO):** T03, Computerwissenschaften, Jakob-Haringer Str. 2.

**Lecture times (PS):** Thursday 14\textsuperscript{00}–16\textsuperscript{00}.

**Venue (PS):** T03, Computerwissenschaften, Jakob-Haringer Str. 2.

**Note** — PS is graded according to continuous-assessment mode!
Electronic Slides and Online Material

In addition to these slides, you are encouraged to consult the WWW home-page of this lecture:

https://www.cosy.sbg.ac.at/~held/teaching/aads/aads.html.

In particular, this WWW page contains up-to-date information on the course, plus links to online notes, slides and (possibly) sample code.
A Few Words of Warning

- I hope that these slides will serve as a practice-minded introduction to various aspects of advanced algorithms and data structures. I would like to warn you explicitly not to regard these slides as the sole source of information on the topics of my course. It may and will happen that I’ll use the lecture for talking about subtle details that need not be covered in these slides! In particular, the slides won’t contain all sample calculations, proofs of theorems, demonstrations of algorithms, or solutions to problems posed during my lecture. That is, by making these slides available to you I do not intend to encourage you to attend the lecture on an irregular basis.

- See also In Praise of Lectures by T.W. Körner.

- A basic knowledge of algorithms, data structures, elementary probability theory, and discrete mathematics, as taught typically in undergraduate courses, should suffice to take this course. It is my sincere intention to start at a suitable hypothetical level of “typical prior undergrad knowledge”. Still, it is obvious that different educational backgrounds will result in different levels of prior knowledge. Hence, you might realize that you do already know some items covered in this course, while you lack a decent understanding of some items which I seem to presuppose. In such a case I do expect you to refresh or fill in those missing items on your own!
Acknowledgments

A small portion of these slides is based on notes and slides produced by students and colleagues — most notably Therese Biedl, Jeff Erickson, Pat Morin’s “Open Data Structures”, Paolo di Stolfo, Peter Palfrader — on topics related to algorithms and data structures. I would like to express my thankfulness to all of them for their help. This revision and extension was carried out by myself, and I am responsible for all errors.

I am also happy to acknowledge that I benefited from material published by colleagues on diverse topics that are partially covered in this lecture. While some of the material used for this lecture was originally presented in traditional-style publications (such as textbooks), some other material has its roots in non-standard publication outlets (such as online documentations, electronic course notes, or user manuals).

Salzburg, August 2019

Martin Held
Legal Fine Print and Disclaimer

To the best of my knowledge, these slides do not violate or infringe upon somebody else’s copyrights. If copyrighted material appears in these slides then it was considered to be available in a non-profit manner and as an educational tool for teaching at an academic institution, within the limits of the “fair use” policy. For copyrighted material we strive to give references to the copyright holders (if known). Of course, any trademarks mentioned in these slides are properties of their respective owners.

Please note that these slides are copyrighted. The copyright holder grants you the right to download and print the slides for your personal use. Any other use, including instructional use at non-profit academic institutions and re-distribution in electronic or printed form of significant portions, beyond the limits of “fair use”, requires the explicit permission of the copyright holder. All rights reserved.

These slides are made available without warrant of any kind, either express or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. In no event shall the copyright holder and/or his respective employer be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, arising out of or in connection with the use of information provided in these slides.
Recommended Textbooks for Background Reading I

P. Brass.  
*Advanced Data Structures.*  
https://doi.org/10.1017/CBO9780511800191  
http://www-cs.ccny.cuny.edu/~peter/dtest.html

P. Morin.  
*Open Data Structures.*  
https://opendatastructures.org/

D. Sheehy.  
*datastructures.*  
https://donsheehy.github.io/datastructures/

J. Erickson.  
*Algorithms.*  
http://jeffe.cs.illinois.edu/teaching/algorithms/

*Introduction to Algorithms.*  
Recommended Textbooks for Background Reading II

R. Sedgewick, K. Wayne
*Algorithms.*
http://algs4.cs.princeton.edu/home/

R. Motwani, P. Raghavan.
*Randomized Algorithms.*

J. Kleinberg, É. Tardos.
*Algorithm Design.*

D.E. Knuth.
Addison-Wesley, 3rd edition, 1997; 978-0201896831.

D.E. Knuth.
# Table of Content

1. Introduction
2. Basics of Algorithm Theory
3. Algorithmic Paradigms
4. Order Statistics, Selection and Sorting
5. Priority Queues
6. Randomized Data Structures for Searching
7. Multi-Dimensional Data and Geometric Queries
8. Hard Problems and Approximation Algorithms
9. Linear and Integer Linear Programming
1 Introduction

- Motivation
- Notation and Terminology
- Math Basics
- Elementary Probability
- Complexity Analysis
**Problem: Euclidean Traveling Salesman Problem (ETSP)**

**Input:** A set $S$ of $n$ points in the Euclidean plane.

**Output:** A cycle of minimal length that starts and ends in one point of $S$ and visits all points of $S$.

Natural strategy to solve an instance of ETSP:

1. Pick a point $p_0 \in S$.
2. Find its nearest neighbor $p' \in S$, move to $p'$, and let $p := p'$.
3. Continue from $p$ to the nearest unvisited neighbor $p' \in S$ of $p$, and let $p := p'$.
4. Repeat the last step until all points have been visited, and return back to $p_0$. 
It’s Obvious!?

- The strategy to always pick the shortest missing link can be seen as a *greedy* strategy. (More on greedy strategies later during this course.)
- It is obvious that this strategy will always solve ETSP, isn’t it?
- Well ... The tour computed need not even be close in length to the optimum tour!
- In the example, the tour computed has length 58, while the optimum tour has length 46!

**Intuition ...**

... is important, but may not replace formal reasoning. Intuition might misguide, and algorithm design without formal reasoning does not make sense.
Will it Terminate?

Caveat

Even seemingly simple algorithms need not be easy to understand and analyze.

```c
void Collatz(int n)
{
    while (n>1) {
        if (ODD(n)) n := 3n + 1;
        else n := n / 2;
    }
    printf("done!\n");
    return;
}
```

- It is not known whether the so-called Collatz $3n + 1$ recursion [Collatz 1937] will terminate for all $n \in \mathbb{N}$.
- Conjecture: It terminates for all $n \in \mathbb{N}$.
- Best case: If $n$ is a power of 2 then it ends in $\log n$ steps. Thus, $\Omega(\log n)$ is a lower bound for the number of executions of the body of the loop.
- Experiments have confirmed the Collatz conjecture up to roughly $5 \cdot 10^{18}$.
Notation

- **Numbers:**
  - The set \{1, 2, 3, \ldots\} of natural numbers is denoted by \( \mathbb{N} \), with \( \mathbb{N}_0 := \mathbb{N} \cup \{0\} \).
  - The set \{2, 3, 5, 7, 11, 13, \ldots\} \subset \mathbb{N} of prime numbers is denoted by \( \mathbb{P} \).
  - The (positive and negative) integers are denoted by \( \mathbb{Z} \).
  - \( \mathbb{Z}_n := \{0, 1, 2, \ldots, n - 1\} \) and \( \mathbb{Z}_n^+ := \{1, 2, \ldots, n - 1\} \) for \( n \in \mathbb{N} \).
  - The reals are denoted by \( \mathbb{R} \); the non-negative reals are denoted by \( \mathbb{R}_0^+ \), and the positive reals by \( \mathbb{R}^+ \).

- **Open or closed intervals** \( I \subset \mathbb{R} \) are denoted using square brackets: e.g., \( I_1 = [a_1, b_1] \) or \( I_2 = [a_2, b_2[ \), with \( a_1, a_2, b_1, b_2 \in \mathbb{R} \), where the right-hand “[” indicates that the value \( b_2 \) is not included in \( I_2 \).

- The set of all elements \( a \in A \) with property \( P(a) \), for some set \( A \) and some predicate \( P \), is denoted by
  \[
  \{ x \in A : P(x) \} \quad \text{or} \quad \{ x : x \in A \land P(x) \}
  \]
  or
  \[
  \{ x \in A \mid P(x) \} \quad \text{or} \quad \{ x : x \in A \land P(x) \}.
  \]

- **Bold capital letters**, such as \( \mathbf{M} \), are used for matrices.

- The set of all (real) \( m \times n \) matrices is denoted by \( M_{m \times n} \).
Notation

Points are denoted by letters written in italics: $p, q$ or, occasionally, $P, Q$. We do not distinguish between a point and its position vector.

The coordinates of a vector are denoted by using indices (or numbers): e.g., $\mathbf{v} = (v_x, v_y)$ for $\mathbf{v} \in \mathbb{R}^2$, or $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ for $\mathbf{v} \in \mathbb{R}^n$.

In order to state $\mathbf{v} \in \mathbb{R}^n$ in vector form we will mix column and row vectors freely unless a specific form is required, such as for matrix multiplication.

The vector dot product of two vectors $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$ is denoted by $\langle \mathbf{v}, \mathbf{w} \rangle$. That is, $\langle \mathbf{v}, \mathbf{w} \rangle = \sum_{i=1}^{n} v_i \cdot w_i$ for $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$.

The vector cross-product (in $\mathbb{R}^3$) is denoted by a cross: $\mathbf{v} \times \mathbf{w}$.

The length of a vector $\mathbf{v}$ is denoted by $\| \mathbf{v} \|$.

The straight-line segment between the points $p$ and $q$ is denoted by $\overline{pq}$.

The supporting line of the points $p$ and $q$ is denoted by $\ell(p, q)$.

The cardinality of a set $A$ is denoted by $|A|$.

Quantifiers: The universal quantifier is denoted by $\forall$, and $\exists$ denotes the existential quantifier.
Terminology

- Unfortunately, the terminology used in textbooks and research papers on algorithms and data structures often lacks a rigorous standardization.
- This comment is particularly true for the underlying graph theory!
- We will rely on the terminology and conventions used in my course on “Discrete Mathematics”.

Advice

Please make sure to familiarize yourself with the terminology and conventions used in "Discrete Mathematics"!
### Logarithms

#### Definition 1 (Logarithm)

The *logarithm* of a positive real number $x \in \mathbb{R}^+$ with respect to a base $b$, which is a positive real number not equal to 1, is the unique solution $y$ of the equation $b^y = x$. It is denoted by $\log_b x$.

- Hence, it is the exponent by which $b$ must be raised to yield $x$.
- Common bases:

$$\log_2 x := \log_2 x \quad \ln x := \log_e x \quad \text{with} \quad e := \lim_{n \to \infty} \left( n + \frac{1}{n} \right) \approx 2.71828 \ldots$$

#### Lemma 2

Let $x, y, p \in \mathbb{R}^+$ and $b \in \mathbb{R}^+ \setminus \{1\}$.

$$\log_b(xy) = \log_b(x) + \log_b(y) \quad \log_b \left( \frac{x}{y} \right) = \log_b(x) - \log_b(y)$$

$$\log_b \left( x^p \right) = p \log_b(x) \quad \log_b \left( \sqrt[p]{x} \right) = \frac{\log_b(x)}{p}$$
Lemma 3 (Change of base)

Let \( x \in \mathbb{R}^+ \) and \( \alpha, \beta \in \mathbb{R}^+ \setminus \{1\} \). Then \( \log_\alpha(x) \) and \( \log_\beta(x) \) differ only by a multiplicative constant:

\[
\log_\alpha(x) = \frac{1}{\log_\beta(\alpha)} \cdot \log_\beta(x)
\]

Convention

In this course, \( \log n \) will always denote the logarithm of \( n \) to the base 2, i.e., \( \log n := \log_2 n \).
Fibonacci Numbers

**Definition 4 (Fibonacci numbers)**

For all \( n \in \mathbb{N}_0 \),

\[
F_n := \begin{cases} 
  n & \text{if } n \leq 1, \\
  F_{n-1} + F_{n-2} & \text{if } n \geq 2.
\end{cases}
\]

<table>
<thead>
<tr>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F_n )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td>89</td>
<td>144</td>
<td>233</td>
<td>377</td>
<td>610</td>
</tr>
</tbody>
</table>

**Lemma 5**

For \( n \in \mathbb{N} \) with \( n \geq 2 \):

\[
F_n = \frac{1}{\sqrt{5}} \cdot \left( \frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \cdot \left( \frac{1 - \sqrt{5}}{2} \right)^n \geq \left( \frac{1 + \sqrt{5}}{2} \right)^{n-2}
\]

- Lots of interesting mathematical properties. For instance,

\[
\lim_{n \to \infty} \frac{F_{n+1}}{F_n} = \phi, \quad \text{where } \phi := \frac{1 + \sqrt{5}}{2} = 1.618 \ldots \text{ is the golden ratio.}
\]
Catalan Numbers

**Definition 6 (Catalan numbers)**

For \( n \in \mathbb{N}_0 \),

\[
C_0 := 1 \quad \text{and} \quad C_{n+1} := \sum_{i=0}^{n} C_i \cdot C_{n-i}.
\]

<table>
<thead>
<tr>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_n )</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>14</td>
<td>42</td>
<td>132</td>
<td>429</td>
<td>1430</td>
<td>4862</td>
<td>16796</td>
<td>58786</td>
</tr>
</tbody>
</table>

**Lemma 7**

For \( n \in \mathbb{N}_0 \),

\[
C_n = \frac{1}{n+1} \sum_{i=0}^{n} \binom{n}{i}^2 = \frac{1}{n+1} \binom{2n}{n} \in \Theta \left( \frac{4^n}{n^{1.5}} \right).
\]
Harmonic Numbers

**Definition 8 (Harmonic numbers)**

For \( n \in \mathbb{N} \),

\[
H_n := 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}.
\]

**Lemma 9**

The sequence \( s: \mathbb{N} \to \mathbb{R} \) with

\[
s_n := H_n - \ln n
\]

is monotonically decreasing and convergent. Its limit is the Euler-Mascheroni constant

\[
\gamma := \lim_{n \to +\infty} (H_n - \ln n) \approx 0.5772\ldots,
\]

and we have

\[
\ln n < H_n - \gamma < \ln(n+1), \quad \text{i.e.} \quad H_n \in \Theta(\ln) = \Theta(\log).
\]
Discrete Probability: Material for Experiments

Basic elementary probability needed . . .

. . . for, e.g., analyzing randomized algorithms and data structures!

Coin:
- A coin has two sides: $H$ (for “head”) or $T$ (for “tail”).

Die:
- A standard die has six sides which are labelled with the numbers 1,2,3,4,5, and 6.
- Rolling a fair die will result in any of these six numbers being up.

Cards:
- A standard 52-card deck of playing cards has 13 hearts (Dt. Herz), 13 diamonds (Dt. Karo), 13 spades (Dt. Pik), and 13 clubs (Dt. Treff).
- Hearts and diamonds are red suits (Dt. Farben); spades and clubs are black suits.
- For each suit, there is a 2, 3, 4, 5, 6, 7, 8, 9, 10, jack, queen, king, and ace. Jacks, queens, and kings are so-called “face” cards.
Discrete Probability

- A trial is one instance of an experiment like rolling a fair die, flipping a coin or pulling a card from decently shuffled deck.

**Definition 10 (Sample space, Dt.: Ergebnisraum)**

A sample space $\Omega$ is a non-empty, finite or countably infinite set. Each element of $\Omega$ is called an outcome (aka elementary event, Dt.: Elementarereignis), and each subset of $\Omega$ is called an event.

**Definition 11 (Probability measure, Dt.: Wahrscheinlichkeit(sfunktion))**

A probability measure $\Pr: \mathcal{P}(\Omega) \rightarrow \mathbb{R}$ is a mapping from the power set $\mathcal{P}(\Omega)$ to $\mathbb{R}$ with the following properties:

- $0 \leq \Pr(A) \leq 1$ for all $A \subseteq \Omega$,
- $\sum_{\omega \in \Omega} \Pr(\omega) = 1$.

- This implies $\Pr \left( \sum_{n \in \mathbb{N}} A_n \right) = \sum_{n \in \mathbb{N}} \Pr(A_n)$ for every sequence $A_1, A_2, \ldots$ of pairwise disjoint sets from $\mathcal{P}(\Omega)$.
Definition 12 (Discrete probability space, Dt.: diskreter Wahrscheinlichkeitsraum)

A *discrete probability space* is a pair \((\Omega, \Pr)\) where \(\Omega\) is a sample space and \(\Pr\) is a probability measure on \(\Omega\).

- The probability of an event \(A \subseteq \Omega\) is defined as the sum of the probabilities of the outcomes of \(A\): \(\Pr(A) := \sum_{\omega \in A} \Pr(\omega)\).
- Other common ways to denote the probability of \(A\) are \(\Pr[A]\) and \(P(A)\) and \(p(A)\).
- In the language of random experiments we understand \(\Pr(A)\) for \(A \subseteq \Omega\) as follows:
  \[
  \Pr(A) = \frac{\text{number of outcomes favorable to } A}{\text{total number of possible outcomes}}
  \]

Definition 13 (Uniform probability space)

A probability space \((\Omega, \Pr)\) is *uniform* if \(\Omega\) is finite and if for every \(\omega \in \Omega\)

\[
\Pr(\omega) = \frac{1}{|\Omega|}.
\]
Discrete Probability

Lemma 14

**Complementarity:** If $A$ is an event that occurs with probability $\Pr(A)$, then $1 - \Pr(A)$ is the probability that $A$ does not occur.

**Sum:** If $A \cap B = \emptyset$ for two events $A$, $B$, i.e., if $A$, $B$ cannot occur simultaneously, then the probability $\Pr(A \cup B)$ that either of them occurs is $\Pr(A) + \Pr(B)$.

Definition 15 (Conditional probability, Dt.: bedingte Wahrscheinlichkeit)

The *conditional probability* of $A$ given $B$, denoted by $\Pr(A \mid B)$, is the probability that the event $A$ occurs given that the event $B$ has occurred:

$$\Pr(A \mid B) = \frac{\Pr(A \cap B)}{\Pr(B)}$$
Discrete Probability

**Definition 16 (Independent Events)**

If \( \Pr(B) > 0 \) then event \( A \) is independent of event \( B \) if and only if

\[
\Pr(A \mid B) = \Pr(A).
\]

**Caveat**

Disjoint events are not independent! If \( A \cap B = \emptyset \), then knowing that event \( B \) happened means that you know that \( A \) cannot happen!

**Lemma 17**

Two events \( A, B \) are independent if and only if either of the following statements is true:

\[
\Pr(A) \cdot \Pr(B) = \Pr(A \cap B) \quad \Pr(A \mid B) = \Pr(A) \quad \Pr(B \mid A) = \Pr(B)
\]

- If any one of these statements is true, then all three statements are true.
Definition 18 (With high probability)

For $n \in \mathbb{N}$, an event $A_n$ occurs with high probability if its probability depends on an integer $n$ and goes to 1 as $n$ goes to infinity.

- Typical example:
  \[
  \Pr(A_n) = \left(1 - \frac{1}{n^c}\right) \quad \text{for some } c \in \mathbb{R}^+.
  \]

- The term “with high probability” is commonly abbreviated as w.h.p. or WHP.
Random Variable

Definition 19 (Random variable, Dt.: Zufallsvariable)

A random variable $X$ on a sample space $\Omega$ is a function $X: \Omega \rightarrow \mathbb{R}$ that maps each outcome of $\Omega$ to a real number. A random variable is discrete if it has a finite or countably infinite set of distinct possible values; it is continuous otherwise. It is called an indicator random variable if $X(\Omega) = \{0, 1\}$.

Misleading terminology!

A random variable is neither “random” nor a “variable”!

- The notation

$$X = a$$

is a frequently used short-hand notation for denoting the set of outcomes $\omega \in \Omega$ such that $X(\omega) = a$. Hence, $X = a$ is an event.
- Similarly for $X \geq a$.

Definition 20 (Independent random variables)

The two random variables $X_1, X_2: \Omega \rightarrow \mathbb{R}$ are independent if for all $x_1, x_2$ the two events $X_1 = x_1$ and $X_2 = x_2$ are independent.
**Probability Distribution**

**Definition 21 (Probability distribution, Dt.: Wahrscheinlichkeitsverteilung)**

For a discrete random variable $X$ on a probability space $(\Omega, \Pr)$, its *probability distribution* is the function $D : \mathbb{R} \rightarrow \mathbb{R}$ with

$$D(x) := \begin{cases} \Pr(X = x) & \text{if } x \in X(\Omega), \\ 0 & \text{if } x \notin X(\Omega). \end{cases}$$

It is *uniform* (Dt.: gleichverteilt) for a finite codomain $X(\Omega)$ if $D(x) = 1/n$ for all $x \in X(\Omega)$, with $n := |X(\Omega)|$.

- The sum of all probabilities contained in a probability distribution needs to equal 1, and each individual probability must be between 0 and 1, inclusive.

**Definition 22 (Cumulative distribution, Dt.: kumulative Wahrscheinlichkeitsverteilung)**

For a discrete random variable $X$ on a probability space $(\Omega, \Pr)$, its *cumulative probability distribution* is the function $CD : X(\Omega) \rightarrow \mathbb{R}$ with

$$CD(x) := \Pr(X \leq x).$$
Expected Value of a Random Variable

**Definition 23 (Expected value, Dt.: Erwartungswert)**

The *expected value*, $\mathbb{E}(X)$, of a discrete random variable $X$ on a probability space $(\Omega, \Pr)$ is defined as

$$\mathbb{E}(X) := \sum_{\omega \in \Omega} X(\omega) \cdot \Pr(\omega),$$

provided that this series converges absolutely.

- That is, the sum must remain finite if all $X(\omega)$ were replaced by their absolute values $|X(\omega)|$.
- The expected value of $X$ can be rewritten as $\mathbb{E}(X) := \sum_{x \in X(\Omega)} x \cdot \Pr(X = x)$.
- Another commonly used term to denote the expected value of $X$ is $\mu_X$. 
Expected Value of a Random Variable

**Lemma 24 (Linearity of expectation)**

Let $a, b, c \in \mathbb{R}$ and two random variables $X, Y$ defined over the same probability space. Then


**Lemma 25 (Markov’s inequality)**

Let $X$ be a non-negative random variable and $a \in \mathbb{R}^+$. Then the probability that $X$ is at least as large as $a$ is at most as large as the expectation of $X$ divided by $a$:

$$\Pr(X \geq a) \leq \frac{E(X)}{a}.$$
Asymptotic Notation: Big-O

Definition 26 (Big-O, Dt.: Groß-O)

Let \( f : \mathbb{N} \rightarrow \mathbb{R}^+ \). Then the set \( O(f) \) is defined as

\[
O(f) := \left\{ g : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c_2 \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 \quad g(n) \leq c_2 \cdot f(n) \right\}.
\]

- Equivalent definition used by some authors:
  \[
  O(f) := \left\{ g : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c_2 \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 \quad \frac{g(n)}{f(n)} \leq c_2 \right\}.
  \]
- Some authors prefer to use the symbol \( \mathcal{O} \) instead of \( O \).
Asymptotic Notation: Big-Omega

Definition 27 (Big-Omega, Dt.: Groß-Omega)

Let \( f : \mathbb{N} \rightarrow \mathbb{R}^+ \). Then the set \( \Omega(f) \) is defined as

\[
\Omega(f) := \left\{ g : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c_1 \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \geq n_0 \; \exists c_1 \cdot f(n) \leq g(n) \right\}.
\]

Equivalently,

\[
\Omega(f) := \left\{ g : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c_1 \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \geq n_0 \; c_1 \leq \frac{g(n)}{f(n)} \right\}.
\]
Asymptotic Notation: Big-Theta

Definition 28 (Big-Theta, Dt.: Groß-Theta)

Let \( f : \mathbb{N} \rightarrow \mathbb{R}^+ \). Then the set \( \Theta(f) \) is defined as

\[
\Theta(f) := \{ g : \mathbb{N} \rightarrow \mathbb{R}^+ | \exists c_1, c_2 \in \mathbb{R}^+ \ \exists n_0 \in \mathbb{N} \ \forall n \geq n_0 \ c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n) \}.
\]

This is equivalent to

\[
c_1 \leq \frac{g(n)}{f(n)} \leq c_2 \quad \text{for all } n \geq n_0
\]
Asymptotic Notation: Small-Oh and Small-Omega

**Definition 29 (Small-Oh, Dt.: Klein-O)**

Let \( f : \mathbb{N} \to \mathbb{R}^+ \). Then the set \( o(f) \) is defined as

\[
o(f) := \{ g : \mathbb{N} \to \mathbb{R}^+ | \forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 \quad g(n) \leq c \cdot f(n) \}.
\]

**Definition 30 (Small-Omega, Dt.: Klein-Omega)**

Let \( f : \mathbb{N} \to \mathbb{R}^+ \). Then the set \( \omega(f) \) is defined as

\[
\omega(f) := \{ g : \mathbb{N} \to \mathbb{R}^+ | \forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 \quad g(n) \geq c \cdot f(n) \}.
\]

- We can extend Defs. 26–30 such that \( \mathbb{N}_0 \) rather than \( \mathbb{N} \) is taken as the domain (Dt.: Definitionsmenge). We can also replace the codomain (Dt.: Zielbereich) \( \mathbb{R}^+ \) by \( \mathbb{R}_{0}^+ \) (or even \( \mathbb{R} \)) provided that all functions are eventually positive.

**Warning**

The use of the equality operator “=” instead of the set operators “\( \in \)” or “\( \subseteq \)” to denote set membership or a subset relation is a common abuse of notation.
Master Theorem

**Theorem 31**

Consider constants $n_0 \in \mathbb{N}$ and $a \in \mathbb{N}$, $b \in \mathbb{R}$ with $b \geq 2$, and a function $f : \mathbb{N} \rightarrow \mathbb{R}^+_0$. Let $T : \mathbb{N} \rightarrow \mathbb{R}^+_0$ be an eventually non-decreasing function such that

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

for all $n \in \mathbb{N}$ with $n \geq n_0$, where we interpret $\frac{n}{b}$ as either $\lceil \frac{n}{b} \rceil$ or $\lfloor \frac{n}{b} \rfloor$.

Then we have

$$T \in \begin{cases} 
\Theta(f) & \text{if } f \in \Omega\left(n^{(\log_b a) + \varepsilon}\right) \text{ for some } \varepsilon \in \mathbb{R}_+, \\
\Theta\left(n^{\log_b a} \log n\right) & \text{if } f \in \Theta\left(n^{\log_b a}\right), \\
\Theta\left(n^{\log_b a}\right) & \text{if } f \in O\left(n^{(\log_b a) - \varepsilon}\right) \text{ for some } \varepsilon \in \mathbb{R}_+. 
\end{cases}$$

- This is a simplified version of the Akra-Bazzi Theorem [Akra&Bazzi 1998].
Basics of Algorithm Theory

- Terminology
- Model of Computation
- Growth Rates
- Reductions
- Proving Lower Bounds
- Amortized Analysis
- Practical Considerations
Problem

A problem is the concise (abstract) description of every permissible input and of the output sought for every (permissible) input.

- E.g., we can specify the sorting problem for (real) numbers as follows:

Problem: SORTING

Input: A sequence of $n$ (real) numbers $(x_1, x_2, \ldots, x_n)$, for some $n \in \mathbb{N}$.

Output: A permutation $\pi \in S_n$ such that $x_{\pi(1)} \leq x_{\pi(2)} \leq \ldots \leq x_{\pi(n)}$.

Problem instance

An instance of a problem is one particular (permissible) input.

- E.g., sorting the five numbers of the sequence $(3, 1, 5, 14, 8)$ forms one instance of the SORTING problem.
- Hence, $n = 5$ and SORTING these numbers requires us to find the permutation

$$
\pi = \begin{pmatrix}
1 & 2 & 3 & 4 & 5 \\
2 & 1 & 3 & 5 & 4
\end{pmatrix}.
$$

- The term “algorithm” is generally assumed to stem from the (Latin transliteration) of the name of a Persian mathematician who was a scholar at the House of Wisdom at Baghdad during the Abbasid Caliphate: Muḥammad ibn Mūsā al-Khwārizmī (ca. 780–850).

- An algorithm may be encoded as a procedure, a formula, a recipe, ...

- Attempts to formalize the concept of an algorithm started with work on the Entscheidungsproblem (posed by Hilbert in 1928). Formalizations include
  - the theory of recursive functions [Gödel, Herbrand, Kleene 1930–1935],
  - lambda calculus [Church 1936],
  - Turing machines [Turing 1936–1939].

- See a textbook on theoretical computer science for formal foundations of “algorithm”.

- In this lecture we will presuppose a general understanding of “algorithm” and use English language, pseudocode or C/C++ as algorithmic notations.
Decision Problem

Definition 32 (Decision Problem; Dt.: Entscheidungsproblem)

A problem is a *decision problem* if the output sought for a particular instance of the problem always is the answer *yes* or *no*.

- Famous decision problem: Boolean satisfiability (SAT).

Problem: SAT

**Input:** A propositional formula $A$.

**Decide:** Is $A$ satisfiable? I.e., does there exist an assignment of truth values to the Boolean variables of $A$ such that $A$ evaluates to true?

- Note that a solution to SAT does not necessarily require us to know suitable truth assignments to the Boolean variables.
- However, if we are given truth assignments for which $A$ is claimed to evaluate to true then this claim is easy to verify.
- We’ll get back to this issue when talking about $NP$-completeness . . .
Decision Problem vs. Computational/Optimization Problems

- Often a decision problem is closely related to an underlying computational/optimization problem.
  E.g., “Sort the numbers $x_1, \ldots, x_n$” versus “Are the numbers $x_1, \ldots, x_n$ sorted?”

**Problem: CHROMATIC NUMBER**

**Input:** An undirected graph $G$.

**Output:** An assignment of colors to the nodes of $G$ such that no neighboring nodes bear the same color and such that a minimum number of colors is used.

**Problem: $k$-COL**

**Input:** An undirected graph $G$ and a constant $k \in \mathbb{N}$ with $k > 3$.

**Decide:** Do $k$ colors suffice to color the nodes of $G$ such that no neighboring nodes bear the same color?
Problem: **EUCLIDEAN TRAVELING SALESMAN PROBLEM (ETSP)**

**Input:** A set $S$ of $n$ points in the Euclidean plane.

**Output:** A cycle of minimal length that starts and ends in one point and visits all points of $S$.

---

Problem: **EUCLIDEAN TRAVELING SALESMAN PROBLEM — DECISION**

**Input:** A set $S$ of $n$ points in the (Euclidean) plane and a constant $c \in \mathbb{R}^+$.  

**Decide:** Does there exist a cycle that starts and ends in one point and visits all points of $S$ such that the length of that cycle is less than $c$?
Complexity of an Algorithm

- Typical kinds of complexities studied:
  - time complexity, i.e., a mathematical assessment or estimation of the running time independent of a particular implementation or platform;
  - space complexity, i.e., a mathematical assessment or estimation of the number of memory units consumed by the algorithm;
  - complexity of the output generated.

**Definition 33 (Worst-Case Complexity, Dt.: Komplexität im schlimmsten Fall)**

A *worst-case complexity* of an algorithm is a function $f : \mathbb{N} \rightarrow \mathbb{R}^+$ that gives an upper bound on the number of elementary operations (memory units, ...) used by an algorithm with respect to the size of its input, for all inputs of the same size.

**Definition 34 (Average-Case Complexity, Dt.: Komplexität im durchschnittl. Fall)**

An *average-case complexity* of an algorithm is a function $g : \mathbb{N} \rightarrow \mathbb{R}^+$ that models the average number of elementary operations (memory units, ...) used by an algorithm with respect to the size of its input.

- So, what does “size of its input” mean? And what are “elementary operations”??
Input Size

Definition 35 (Input Size, Dt.: Eingabegröße)

The size of the input of an algorithm is a quantity that measures the number of input items relevant for elementary operations of the algorithm.

- For most problems the choice of an appropriate measure of the input size will be fairly obvious.
- E.g., for sorting a typical measure of the input size will be the number of records to be sorted (if constant memory and comparison time per record may be assumed).
- If we are to check for intersections among line segments then it seems natural to take the number of line segments as input size.
- A graphics rendering application may want to consider the number of triangles to be rendered as input size.
Input Size — It Does Matter!

**Problem: PRIME**

**Input:** A natural number $n$ with $n > 1$.

**Decide:** Is $n$ prime? I.e., can $n$ be divided only by 1 and by itself?

```java
boolean IsPrime(int n) {
    for (j:=2; j<=sqrt(n); j++) {
        if ( (n mod j) == 0 ) return false;
    }
    return true;
}
```

**Complexity:**
- The body of the loop is executed $O(\sqrt{n})$ times.
- If the operation $(n \ mod \ j)$ can be implemented to run in $O(n)$ time, then this algorithm solves problem PRIME in $O(n\sqrt{n})$ steps!
However: What is the input size? Does the description of a number $n$ really require $O(n)$ characters?

- In the dual system: $\text{SIZE}_2(1000) \approx 10$.
- In the decimal system: $\text{SIZE}_{10}(1000) = 4$.
- Thus, in the dual system, an input of size $k$ results in $O((2^k)^{3/2})$ many steps being carried out by our simple algorithm!
- Note: The latter bound is exponential in $k$!
Model of Computation

- We continue with (informal!) definitions that pertain to the complexity analysis of algorithms.

Definition 36 (Elementary Operation, Dt.: Elementaroperation)

An *elementary operation* is an operation whose running time is assumed not to depend on the specific values of its operands.

- E.g., the time taken by the comparison of two floating-point numbers is frequently assumed to be constant.
- Still, what constitutes an elementary operation depends on the model of computation.

Definition 37 (Model of Computation)

A *model of computation* specifies the elementary operations that may be executed, together with their respective costs.
Model of Computation

- Purely theoretical point of view: Turing Machine (TM) model.
- This is the model to use when talking about theoretical issues like \( \mathcal{NP} \)-completeness!
- But the TM model is cumbersome to use for analyzing actual complexities and, thus, is impractical for most “real” applications . . .
- Hence several alternative models have been proposed, e.g.:
  - Random Access Machine (RAM) model,
  - Real RAM model,
  - Algebraic Decision/Computation Tree (ADT/ACT) model,
  - Blum-Shub-Smale model.

Warning

While all these models are “good enough from a practical point of view” to shed some light on the complexity of an algorithm or a problem, they do differ in detail. Different models of computation are not equally powerful, and complexity results need not transfer readily from one model to another model.

- Consult a textbook on theoretical computer science for details . . .
Definition 38 (Random Access Machine, Dt.: Registermaschine)

- Every memory location can hold a single integer number.
- The following operations are available at unit cost:
  1. Reading and writing (i.e., file I/O) of a number;
  2. The three standard arithmetic operations: $+,-,\times$;
  3. Comparisons between two integers: $<,\leq,=,\neq,\geq,>;$
  4. Indirect addressing of memory.

- Aka: Integer RAM.
- The RAM model is not very realistic. E.g., it allows solving PRIME in $O(n \log \log n)$ time, using the sieve of Eratosthenes.
- What about geometry and graphics? Obviously, the RAM model falls short when talking about, say, the intersection of two polynomial curves.
Real RAM

**Definition 39 (Real Random Access Machine (Real RAM), Shamos 1978)**

- Every memory location can hold a single real(!) number.
- The following operations are available at unit cost:
  1. Reading and writing (i.e., file I/O) of a real number;
  2. The four standard arithmetic operations and square roots: $+, -, \times, \div, \sqrt{}$;
  3. Comparisons between two reals: $<, \leq, =, \neq, \geq, >$;
  4. Indirect addressing of memory (for integer addresses only!).

- Real RAM is a widely used model of computation, e.g., in geometric modeling and computational geometry.
- It is unrealistic, though, since just one real number can encode an infinite amount of information!
- The Real RAM is truly more powerful than the RAM! E.g., given two finite sets $S$ and $T$ of positive integers, decide whether

$$\sum_{s \in S} \sqrt{s} > \sum_{t \in T} \sqrt{t}.$$
Real RAM and Floor Function

- A real RAM does not allow to convert from real variables to integer variables. In particular, it does not allow the use of the floor function $\lfloor \cdot \rfloor$.

**Problem: MaxGap**

**Input:** A set $S$ of $n > 2$ distinct (unsorted!) real numbers.

**Output:** The maximum gap $\delta$ among the sorted numbers of $S$.

$$\delta := \max \{ y - x : x, y \in S \land x < y \land \neg (\exists z \in S : x < z < y) \}. $$

- MaxGap has an $\Omega(n \log n)$ lower bound in the ACT model.

**Theorem 40 (Gonzalez 1975)**

Adding the floor function $\lfloor \cdot \rfloor$ to the Real RAM allows to solve MaxGap in $\Theta(n)$ time.

- [Schönhage 1979]: Adding the floor function allows to solve $NP$-hard problems in polynomial time!
Proof of Theorem 40:

- Let $S := \{x_1, x_2, \ldots, x_n\} \subset \mathbb{R}$.
- $x_{\text{min}} := \min(S)$ and $x_{\text{max}} := \max(S)$.
- Divide the interval $[x_{\text{min}}, x_{\text{max}}]$ into $(n - 1)$ “buckets” of equal size
  $$\delta := \frac{x_{\text{max}} - x_{\text{min}}}{(n-1)}.$$
- Assign the $n - 2$ numbers of $S \setminus \{x_{\text{min}}, x_{\text{max}}\}$ to the buckets: The number $x_i$ is assigned to the $k$-th bucket if and only if
  $$\left\lfloor \frac{(x_i - x_{\text{min}})}{\delta} \right\rfloor = k - 1.$$
- For each non-empty bucket $B_k$ compute $x_{\text{min}}^k$ and $x_{\text{max}}^k$ among the numbers assigned to $B_k$.
- Since we assigned $n - 2$ numbers to $n - 1$ buckets, the Pigeonhole Principle implies that at least one bucket stays empty. Hence, the maximum gap cannot occur inside of a bucket.
- Form a sorted list of all ordered minima and maxima (of all non-empty buckets):
  $$M := ((x_{\text{min}}, x_{\text{min}}), (x_{\text{min}}^1, x_{\text{max}}^1), (x_{\text{min}}^2, x_{\text{max}}^2), \ldots, (x_{\text{min}}^{n-1}, x_{\text{max}}^{n-1}), (x_{\text{max}}, x_{\text{max}})).$$
- Scan $M$ to find the maximum distance between a pair of consecutive minima and maxima. This number is the maximum gap.
- Obviously, this algorithm runs in $O(n)$ time.
Algebraic Computation Tree

Definition 41 (Algebraic computation tree, Dt.: algebr. Berechnungsbaum)

An algebraic computation tree with input \((x_1, x_2, \ldots, x_n) \in \mathbb{R}^n\) solves a decision problem \(P\) if it is a finite rooted tree with at most two children per node and two types of internal nodes:

**Computation:** A computation node \(v\) has a value \(f_v\) determined by one of the following instructions:

\[
\begin{align*}
    f_v &= f_u \circ f_w \quad \text{or} \quad f_v = \sqrt{f_u}
\end{align*}
\]

where \(\circ \in \{+, -, \cdot, /\}\) and \(f_u, f_w\) are values associated with ancestors of \(v\), input variables or arbitrary real constants.

**Branch:** A branch node \(v\) has two children and contains one of the predicates

\[
\begin{align*}
    f_u &> 0 \\
    f_u &\geq 0 \\
    f_u &= 0
\end{align*}
\]

where \(f_u\) is a value associated with an ancestor of \(v\) or an input variable.

Every leaf node is associated with Yes and No, depending on the correct answer for every \((x_1, x_2, \ldots, x_n)\) relative to \(P\).
Membership Set

- Of course, we require that no computation node leads to a division by zero or to taking the square root of a negative number.

**Definition 42 (Membership set)**

For a decision problem $P$ with input variables $x_1, x_2 \ldots, x_n \in \mathbb{R}$ we define $W_P$ as the set of points in $\mathbb{R}^n$ for which the answer to the decision problem is Yes:

$$W_P := \{(u_1, \ldots, u_n) \in \mathbb{R}^n : u_1, u_2 \ldots, u_n \text{ yield “Yes” for } P\}.$$  

The set $W_P$ is called the *membership set* of $P$.
Also: $\overline{W_P} := \mathbb{R}^n \setminus W_P$.

- Thus, $\overline{W_P}$ contains the points in $\mathbb{R}^n$ for which the answer is No.

**Definition 43**

For a decision problem $P$ with input $x_1, x_2 \ldots, x_n \in \mathbb{R}$ and membership set $W_P$ we denote the number of disjoint connected components of $W_P$ by $\#(W_P)$, and the number of disjoint connected components of $\overline{W_P}$ by $\#(\overline{W_P})$. 
Theorem 44

If we exclude all intermediate nodes which correspond to additions, subtractions and multiplications by constants then we get for the height $h$ of an algebraic computation tree that solves a decision problem $P$:

$$h = \Omega(\log(\#(W_P) + \#(\overline{W_P})) - n).$$

- Theorem 44 is a consequence of a clever adaption by Steele&Yao [1982] and Ben-Or [1983] of a classical result in algebraic geometry obtained independently by Petrovskii&Oleinik [1952], Milnor [1964] and Thom [1965].

- For fixed-dimensional input, the real RAM model and the ACT model are equivalent. (Otherwise ACT is more powerful since a tree of polynomial depth can be of exponential size.)

- [Blum&Shub&Smale 1989]: Similar to real RAM.
Logarithmic Growth

- Recall Lemma 3:
  \[
  \log_\alpha(n) = \frac{1}{\log_\beta(\alpha)} \cdot \log_\beta(n) \quad \text{for all } \alpha, \beta \in \mathbb{R}^+ \setminus \{1\} \text{ and all } n \in \mathbb{N}.
  \]
  Hence,
  \[
  \Theta(\log_\alpha n) = \Theta(\log_\beta n) \quad \text{for all } \alpha, \beta \in \mathbb{R}^+ \setminus \{1\}.
  \]

- Note that Stirling’s formula asserts
  \[
  n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n, \quad \text{thus, } \Theta(\log n!) = \Theta(n \log n).
  \]

- Recall Lemma 9. Since
  \[
  \lim_{n \to +\infty} (H_n - \ln n) = \gamma,
  \]
  we know that
  \[
  \sum_{k=1}^{n} \frac{1}{k} = \Theta(\ln n) = \Theta(\log n).
  \]
Logarithmic Growth and US Federal Fraud Sentencing

Courts throughout the USA use standardized guidelines in an attempt to standardize criminal sentences. For fraud (part of) the table used for increasing the punishment level is shown below. (The punishment level corresponds roughly to the amount of time served.)

<table>
<thead>
<tr>
<th>Financial damage</th>
<th>increase of level</th>
</tr>
</thead>
<tbody>
<tr>
<td>more than $5000</td>
<td>add 2</td>
</tr>
<tr>
<td>more than $10000</td>
<td>add 4</td>
</tr>
<tr>
<td>more than $30000</td>
<td>add 6</td>
</tr>
<tr>
<td>more than $70000</td>
<td>add 8</td>
</tr>
<tr>
<td>more than $120000</td>
<td>add 10</td>
</tr>
<tr>
<td>more than $200000</td>
<td>add 12</td>
</tr>
<tr>
<td>more than $400000</td>
<td>add 14</td>
</tr>
<tr>
<td>more than $1000000</td>
<td>add 16</td>
</tr>
<tr>
<td>more than $2500000</td>
<td>add 18</td>
</tr>
<tr>
<td>more than $7000000</td>
<td>add 20</td>
</tr>
<tr>
<td>more than $20000000</td>
<td>add 22</td>
</tr>
<tr>
<td>more than $50000000</td>
<td>add 24</td>
</tr>
<tr>
<td>more than $100000000</td>
<td>add 26</td>
</tr>
<tr>
<td>more than $200000000</td>
<td>add 28</td>
</tr>
<tr>
<td>more than $400000000</td>
<td>add 30</td>
</tr>
</tbody>
</table>

Note the roughly logarithmic growth of the punishment level relative to the money stolen!

Lesson learned

Quoting S. Skiena (“Algorithm Design Manual”): If you are gonna do the crime, make it worth the time!
Inverse Ackermann

- \(O(\alpha)\) ........ inverse Ackermann:

\[
A(m, n) := \begin{cases} 
n + 1 & \text{if } m = 0, \\
A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0, \\
A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0.
\end{cases}
\]

- The Ackermann function \(A(m, n)\) grows extremely rapidly. E.g.,
  \(A(4, 3) = 2^{65536} - 3\).
- Hence, the inverse Ackermann function
  \[
  \alpha(n) := \min\{m \in \mathbb{N} : A(m, m) \geq n\}
  \]
  grows extremely slowly; it is at most four for any input of practical relevance.
- But it does grow unboundedly as \(n\) grows, and we have \(1 \in o(\alpha)!\)

- Real-world occurrence of \(O(\alpha)\): Combinatorial complexity of lower envelope of \(n\) line segments.
Consider a set of \( n \) real-valued functions \( f_1, f_2, \ldots, f_n \) over the same domain. Their \textit{lower envelope} is the function \( f_{\ell} \) given by the pointwise minimum of \( f_1, f_2, \ldots, f_n \):

\[
f_{\ell}(x) := \min\{f_i(x): 1 \leq i \leq n\}.
\]

The projection of the lower envelope onto the \( x \)-axis gives a sequence of intervals, and the theory of Davenport-Schinzel sequences implies the following result [Sharir\&Agarwal 1995]: The lower envelope of \( n \) line segments contains at most \( \Theta(n \alpha(n)) \) segments and vertices — and this bound is tight!
Log-star

- $O(\log^* n)$ ........ *log-star* or *iterated logarithm*:

$$\log^* x := \begin{cases} 
0 & \text{if } x \leq 1, \\
1 + \log^*(\log x) & \text{if } x > 1.
\end{cases}$$

- E.g., for the binary logarithm, $\log 2^{16} = 16$ but (since $2^{16} = 2^{2^{2^{2}}}$)

$$\log^* 2^{16} \overset{\text{def}}{=} 1 + \log^* 2^{2^{2}} = 2 + \log^* 2^{2} = 3 + \log^* 2^{1} = 4 + \log^* 1 = 4.$$ 

- Log-star also grows very slowly; it is less than 6 for any input of practical relevance. (Note: $((2^{16})^{2})^{2} = 18446744073709551616$.)
- However, we have $\alpha \in o(\log^*)$!
- Log-star shows up in the complexity bound of Fürer’s algorithm [2007] for multiplying large integers: If $n$ denotes the total number of bits of the two input numbers then an optimized version of his algorithm runs in time $O(n \log n 2^{3 \log^* n})$ [Harvey et al. 2014]. For truly large values of $n$ this is slightly better then the $O(n \log n \log \log n)$ bound of the Schönhage-Strassen algorithm, which is based on Fast Fourier Transform (FFT).
The Expression $k + \varepsilon$

- A term of the form $k + \varepsilon$ means that there is some additive positive constant $\varepsilon \in \mathbb{R}^+$ that needs to be added to $k$.
- The constant $\varepsilon$ may be regarded as arbitrarily small but it will never equal zero.
- Suppose that some algorithm runs in $2^c n^{1+1/c}$ time, where $c \in \mathbb{R}^+$ is a user-chosen constant:
  - For $c := 2$, the complexity term equals $4n^{3/2}$, which is in $O(n^{3/2})$.
  - For $c := 9$, the complexity term equals $2^9 n^{10/9}$, which is in $O(n^{10/9})$.
  - It is easy to see that $1 + 1/c$ approaches 1 as $c$ approaches infinity.
  - However, $c$ cannot be set to infinity (or made arbitrarily large) since then the $2^c$ term would dominate the complexity of our algorithm.
- Hence, the best possible way to express this complexity is to write $O(n^{1+\varepsilon})$. 
Reduction of a Problem: Humorous View of Reductions

Difference between a mathematician and an engineer?

One can perform the following experiment to tell the difference between a mathematician (or theoretical computer scientist) and an engineer:

1. Put an empty kettle in the middle of the kitchen floor and tell your subjects to boil some water.
   - The engineer will fill the kettle with water, put it on the (electric) stove, and turn the electricity on.
   - The mathematician will proceed similarly.

2. Next, put the kettle already filled with water on the stove, and ask the subjects to boil the water.
   - The engineer will turn the electricity on and boil the water.
   - The mathematician will empty the kettle and put it in the middle of the kitchen floor — and claim the problem to be solved by having it reduced to a problem whose solution is already known!
Reduction of a Problem

Definition 46 (Reduction)
A problem $\mathcal{A}$ can be reduced (or transformed) to a problem $\mathcal{B}$ if

1. every instance $A$ of $\mathcal{A}$ can be converted to an instance $B$ of $\mathcal{B}$,
2. a solution $S$ for $B$ can be computed, and
3. $S$ can be transformed back into a correct solution for $A$.

\[ \mathcal{A} \xrightarrow{\tau(|A|)} \mathcal{B} \]

Definition 47
A problem $\mathcal{A}$ is $\tau$-reducible to $\mathcal{B}$, denoted by $\mathcal{A} \leq_\tau \mathcal{B}$, if

1. $\mathcal{A}$ can be reduced to $\mathcal{B}$,
2. for any instance $A$ of $\mathcal{A}$, steps 1 and 3 of the reduction can be carried out in at most $\tau(|A|)$ time, where $|A|$ denotes the input size of $A$.

\[ \mathcal{A} \xrightarrow{\tau(|A|)} \mathcal{B} \]
Transfer of Complexity Bounds

Lemma 48 (Upper bound via reduction)
Suppose that $\mathcal{A}$ is $\tau$-reducible to $\mathcal{B}$ such that the order of the input size is preserved. If problem $\mathcal{B}$ can be solved in $O(T)$ time, then $\mathcal{A}$ can be solved in at most $O(T + \tau)$ time.

Lemma 49 (Lower bound via reduction)
Suppose that $\mathcal{A}$ is $\tau$-reducible to $\mathcal{B}$ such that the order of the input size is preserved. If problem $\mathcal{A}$ is known to require $\Omega(T)$ time, then $\mathcal{B}$ requires at least $\Omega(T - \tau)$ time.
Transfer of Time Bounds: **SORTING** and **ELEMENTUNIQUENESS**

**Problem: ELEMENTUNIQUENESS**

**Input:** A set $S$ of $n$ real numbers $x_1, x_2, \ldots, x_n$.

**Decide:** Are any two numbers of $S$ equal?

- Obviously, after sorting $x_1, x_2, \ldots, x_n$ we can solve ELEMENTUNIQUENESS in $O(n)$ time.
- Hence, reduction yields the following result:

**Lemma 50**

- **ELEMENTUNIQUENESS** can be solved in time $O(f) + O(n)$ if we can sort $n$ numbers in $O(f)$ time, for some $f : \mathbb{N} \to \mathbb{R}^+$.
- **SORTING** requires $\Omega(f)$ time if **ELEMENTUNIQUENESS** requires $\Omega(f)$ time, for some $f : \mathbb{N} \to \mathbb{R}^+$ with $\lambda n.n \in o(f)$. 
Problem: **CLOSESTPAIR**

**Input:** A set $S$ of $n$ points in the Euclidean plane.

**Output:** Those two points of $S$ whose mutual distance is minimum among all pairs of points of $S$.

- We allow points to coincide but still expect them to be distinguishable by some additional data associated with each point. E.g., by means of their indices.
Transfer of Time Bounds: **CLOSESTPAIR** and **ELEMENTUNIQUENESS**

### Lemma 51

1. **ELEMENTUNIQUENESS** can be solved in $O(f) + O(n)$ time if **CLOSESTPAIR** can be solved in $O(f)$ time.

2. **CLOSESTPAIR** requires $\Omega(f)$ time if **ELEMENTUNIQUENESS** requires $\Omega(f)$ time (and if $\lambda n.n \in o(f)$).

**Proof:**

- We use reduction, by reducing **ELEMENTUNIQUENESS** to **CLOSESTPAIR**.
- Let $S' := \{x_1, x_2, \ldots, x_n\} \subset \mathbb{R}$ be an instance of **ELEMENTUNIQUENESS**.
- We transform $S'$ into an instance of **CLOSESTPAIR** by mapping each real number $x_i$ to the point $(x_i, 0) \in \mathbb{R}^2$. All points of the resulting set $S$ of points lie on the x-axis.
- Obviously, all elements of $S'$ are unique if and only if the closest pair of $S$ has a non-zero distance.
- It is also obvious that these transformations are carried out in $O(n)$ time.
- Hence, we get a lower bound on the time complexity of **CLOSESTPAIR** and an upper bound on the time complexity of **ELEMENTUNIQUENESS**.
Transfer of Time Bounds: SORTING and CONVEXHULL

Problem: CONVEXHULL

**Input:** A set $S$ of $n$ points in the Euclidean plane $\mathbb{R}^2$.

**Output:** The convex hull $CH(S)$, i.e., the smallest convex super set of $S$.

Definition 52 (Convex set)

A set $X \subset \mathbb{R}^2$ is **convex** if for every pair of points $p, q \in X$ also the line segment $\overline{pq}$ is contained in $X$.
Lemma 53

\textsc{ConvexHull} requires $\Omega(f)$ time if \textsc{Sorting} requires $\Omega(f)$ time, for some $f : \mathbb{N} \rightarrow \mathbb{R}^+$ with $\lambda n. n \in o(f)$.

- Suppose that the instance of \textsc{Sorting} is the set $S' := \{x_1, x_2, ..., x_n\} \subset \mathbb{R}$.
- We transform $S'$ into an instance of \textsc{ConvexHull} by mapping each real number $x_i$ to the point $(x_i, x_i^2)$. All points of the resulting set $S$ of points lie on the parabola $y = x^2$.
- One pass through $CH(S)$ will find the smallest element. The sorted numbers can be obtained by a second pass through this list, at a total extra cost of $O(n)$ time.
Lower Bound for ELEMENTUNIQUENESS

- What is a lower bound on the number of comparisons for ELEMENTUNIQUENESS?
- We investigate our problem for \( n = 3 \):

\[
\begin{align*}
    x_1 &< x_2 < x_3 & x_2 &< x_3 < x_1 \\
    x_1 &< x_3 < x_2 & x_3 &< x_1 < x_2 \\
    x_2 &< x_1 < x_3 & x_3 &< x_2 < x_1 \\
\end{align*}
\]

If and only if one of these inequalities is true, then all numbers are different, and the answer to our decision problem is No.

- We define the subset \( \overline{W_P} \) of \( \mathbb{R}^3 \) for which the answer is No:

\[
\overline{W_P} := \bigcup_{\pi \in S_3} \overline{W_\pi}
\]

with

\[
\overline{W_\pi} := \{(x_1, x_2, x_3) \in \mathbb{R}^3 : x_{\pi(1)} < x_{\pi(2)} < x_{\pi(3)}\}.
\]

- We get \( \#(\overline{W_P}) = 6 \) because each permutation \( \pi \) results in its own connected component (that is disjoint from all other components of \( \overline{W_P} \)).
Lower Bound for \textsc{ElementUniqueness}

- For $\mathbb{R}^n$ we have $\#(W_P) = n!$:
  - Let $\pi, \sigma \in S_n$ with $\pi \neq \sigma$. For $1 \leq i, j \leq n$ we define
    \[ f_{ij}(x_1, x_2, \ldots, x_n) := x_i - x_j. \]
  - All these functions are continuous and have a constant sign on $W_\pi$ and $W_\sigma$.
  - Since $\pi \neq \sigma$, there exist $i \neq j$ such that
    \[ f_{ij}(p) > 0 \quad \text{for all } p \in W_\pi \quad \text{but} \quad f_{ij}(p) < 0 \quad \text{for all } p \in W_\sigma. \]
  - By the intermediate value theorem of calculus, any path from a point in $W_\pi$ to a point in $W_\sigma$ must pass through a point $q$ where $f_{ij}(q) = 0$.
  - But $q \notin W_P$.
  - Hence, $W_\pi$ and $W_\sigma$ lie in two different connected components if $\pi \neq \sigma$.
  - Since $|S_n| = n!$, we know that $\#(W_P) \geq n!$.

- Based on Theorem 44, we conclude that the height $h$ of an ACT (or ADT) is
  \[ h = \Omega(\log(n!) - n), \quad \text{i.e., that} \quad \Omega(\log n!) = \Omega(n \log n) \]

comparisons are necessary to solve \textsc{ElementUniqueness} in any ACT (and in any ADT of fixed maximum degree) for $n$ input numbers.
Lower Bound for \textsc{ElementUniqueness}

\textbf{Theorem 54}

A (comparison-based) solution of \textsc{ElementUniqueness} for \(n\) real numbers requires \(\Omega(n \log n)\) comparisons.

\textbf{Corollary 55}

A (comparison-based) \textsc{Sorting} of \(n\) real numbers requires \(\Omega(n \log n)\) comparisons.

- Comparison-based sorting means that the sorted order is achieved only by using comparisons among the input elements (relative to a total order on them).

\textbf{Corollary 56}

A solution to \textsc{ConvexHull} for \(n\) points requires \(\Omega(n \log n)\) time in the ACT model in the worst case.

\textbf{Corollary 57}

A solution to \textsc{ClosestPair} for \(n\) points requires \(\Omega(n \log n)\) time in the ACT model in the worst case.
Lower Bounds for \textsc{MaxGap} and \textsc{UniformGap}

**Problem: \textsc{MaxGap}**

**Input:** A set $S$ of $n > 2$ distinct (unsorted!) real numbers.

**Output:** The maximum gap $\delta$ among the sorted numbers of $S$.

$$\delta := \max\{y - x : x, y \in S \land x < y \land \neg(\exists z \in S : x < z < y)\}.$$ 

**Problem: \textsc{UniformGap}**

**Input:** A set $S := \{x_1, x_2, \ldots, x_n\}$ of $n > 2$ distinct (unsorted!) real numbers and $\delta \in \mathbb{R}^+$. 

**Decide:** Is the gap between the sorted numbers of $S$ uniformly $\delta$?

$$\exists \pi \in S_n \ x_{\pi(i+1)} = x_{\pi(i)} + \delta \text{ for all } i \in \{1, 2 \ldots, n - 1\}?$$

**Theorem 58**

A solution to \textsc{UniformGap} for $n$ numbers requires $\Omega(n \log n)$ time in the ACT model.
Corollary 59

A solution to MAXGAP for \( n \) numbers requires \( \Omega(n \log n) \) time in the ACT model.

Proof:

- We reduce UNIFORMGAP in linear time to MAXGAP as follows.
- Given an instance \( \{x_1, x_2, \ldots, x_n\} \) and \( \delta \) of the UNIFORMGAP problem, we first use a MAXGAP algorithm to compute the maximum gap \( \Delta \) of the \( n \) numbers.
- If \( \Delta \neq \delta \) then the answer is No.
- Otherwise, compute \( x_{\text{min}} := \min\{x_1, x_2, \ldots, x_n\} \) and \( x_{\text{max}} := \max\{x_1, x_2, \ldots, x_n\} \), and check whether \( x_{\text{max}} - x_{\text{min}} = (n - 1) \cdot \delta \).
- It is obvious that this algorithm solves the UNIFORMGAP problem, and that it can be run in time \( O(n) \) plus the time consumed by the MAXGAP algorithm.
- Thus, an \( \Omega(n \log n) \) bound is established for MAXGAP.
Adversary Strategy

- Can we come up with a “model” of the worst case?
- For some problems the answer is “yes”, by employing an adversary strategy.
- Two players, $A$ and $B$, play the following game: $A$ thinks of $n$ distinct real numbers, and $B$ tries to sort those numbers by comparing pairs of two numbers.
- Of course, $B$ does not know (the order of) $A$’s numbers.
- Comparisons: $B$ is allowed to ask questions of the form “Is your third number greater than your fifth number?”
- No cheating! $A$ has to answer truthfully and consistently.
- Note, however, that $A$ can replace his originally chosen $n$ numbers by a new $n$-tuple of numbers, at any time during the game, provided that the new numbers are consistent with the answers that $A$ has given so far.
- In particular, if this does not contradict answers given so far, then $A$ can re-order his numbers at any time during the game at his discretion.
- What is a lower bound on the number of comparisons that $A$ can force $B$ to make?
- Player $A$ uses an adversary strategy to prove that $\Omega(n \log n)$ constitutes a lower bound for the number of comparisons which $B$ has to make in the worst case, i.e., to the number of steps that it takes $B$ to sort those $n$ numbers.
- There are $n!$ different permutations. Thus, player $B$ (sorting algorithm) must decide among $n!$ different sequences of comparisons to identify the order of the numbers.
Adversary Strategy: Lower Bound for Sorting

- We assume that \( A \) stores the \( n \) numbers in an array \( a[1, \ldots, n] \), and that \( B \) will sort the numbers by comparing some element \( a[i] \) to some other element \( a[j] \), i.e., by asking \( A \) whether \( a[i] < a[j] \).

- Since the adversary \( A \) is allowed to pick the input, the adversary \( A \) keeps a set \( S \) of permutations that are consistent with the comparisons \( B \) has made so far.

- The answer of \( A \) to a comparison “Is \( a[i] < a[j] \)?” is chosen as follows:
  - Let \( Y \subset S \) be those permutations that have remained in \( S \) and that are also consistent with \( a[i] < a[j] \).
  - Furthermore, \( N := S \setminus Y \).
  - If \( |Y| \geq |N| \) then the adversary \( A \) prefers to answer “yes” and then replaces \( S \) by \( Y \).
  - Otherwise, “no” is answered and \( S \) is replaced by \( N \).

- This strategy allows \( A \) to keep at least half of the permutations after every comparison of the algorithm \( B \).

- Player \( B \) cannot declare the order of the numbers to be known (and, thus, the numbers to be sorted) as long as \( |S| > 1 \).

- Thus, \( B \) needs at least \( \Omega(\log(n!)) = \Omega(n \log n) \) comparisons, which establishes the lower bound sought.
Amortized Analysis: Motivation

- Amortized analysis is a worst-case analysis of a sequence of different operations performed on a datastructure or by an algorithm.
- It is applied if a costly operation cannot occur for a series of operations in a row.
- With traditional worst-case analysis, the resulting bound on the running time of such a sequence of operations is too pessimistic if the execution of a costly operation only happens after many cheap operations have already been carried out.
- The goal of amortized analysis is to obtain a bound on the overall cost of a sequence of operations or the average cost per operation in the sequence which is tighter than what can be obtained by separately analyzing each operation in the sequence.
- Introduced in the mid 1970s to early 1980s, and popularized by Tarjan in “Amortized Computational Complexity” [Tarjan 1985].
- Finance: Amortization refers to paying off a debt by smaller payments made over time.
Amortized Analysis: Dynamic Array

- A *dynamic array* is a data structure of variable size that allows to store and retrieve elements in a random-access mode.

- Elements can also be inserted at and deleted from the end of the array.

- To be distinguished from a dynamically allocated array, which is an array whose size is fixed at the time when the array is allocated.

- Simple realization of a dynamic array: Use a dynamically allocated array of fixed size and reallocate whenever needed to increase (or decrease) the capacity of the array.
  - Size: Number of contiguous elements stored in the dynamic array.
  - Capacity: Physical size of the underlying fixed-sized array.

  Insertion of an element at the end of the array:
  - Constant-time operation if the size is less than the capacity.
  - Costly if the dynamic array needs to be resized since this involves allocating a new underlying array and copying each element from the original array.

- How shall we resize the dynamic array?
Amortized Analysis: Dynamic Array

- Suppose that the initial capacity of the array is 1.
- Simple strategy: We increase the capacity by one whenever the size of the array gets larger than its capacity.
- In the worst case, a sequence of $n$ array operations consists of only insertions at the end of the array, at a cost of $k$ for the insertion of the $k$-th element into an array of size $k-1$.
- Hence, we get $1 + 2 + \ldots + n = \frac{n(n+1)}{2} \in O(n^2)$ as total worst-case complexity for $n$ insert operations, i.e., $O(n)$ per operation.
- Can we do better?

```c
AddAtEndOfArray(dynamicArray A, element e) {
    if (A.size == A.capacity) {
        A.capacity += 1;
        copy contents of A to new memory location;
    }
    A[A.size] = e;
    A.size += 1;
}
```
Amortized Analysis: Dynamic Array

- To avoid the costs of frequent resizing we expand the array by a constant factor $\alpha$ whenever we run out of space. E.g., we can double its capacity.
- Amortized analysis allows to show that the (amortized) cost per array operation is reduced to $O(1)$.
- Amortized constant cost per operation is achieved for any growth factor $\alpha > 1$.
  - C++: `std::vector` with $\alpha = 2$ for GCC 5.2 and $\alpha = \frac{3}{2}$ for Microsoft VC++.
  - Java: `ArrayList` with $\alpha = \frac{3}{2}$.
- The best value for the growth factor is a topic of frequent discussions.
- Could also shrink the array if its size falls below some percentage of its capacity.

```c
InsertIntoArray(dynamicArray A, element e) {
    if (A.size == A.capacity) {
        A.capacity *= 2;
        copy contents to new memory location;
    }
    A[A.size] = e;
    A.size += 1;
}
```
Amortized Analysis: Basics

**Definition 60 (Amortized cost)**

The *amortized cost* of one operation out of a sequence of \( n \) operations (for some \( n \in \mathbb{N} \)) is the total (worst-case) cost for all operations divided by \( n \).

- **Amortized analysis**
  - does not depend on a particularly “good” sequence of operations;
  - considers arbitrary sequences, and, in particular, worst-case sequences;
  - gives genuine upper bounds: the amortized cost per operation times the number of operations yields a worst-case bound on the total complexity of any permissible sequence of those operations;
  - guarantees the average performance of each operation in the worst case;
  - does not involve probabilities;
  - averages over time.

- **Average-case analysis**
  - averages over the input;
  - typically depends on assumptions on probability distributions to obtain an *estimated* cost per operation.
Amortized Analysis: Basics

Warning

Even if the amortized cost of one operation is $O(1)$, the worst-case cost of one particular operation may be substantially greater!

- Hence, studying amortized costs might not be good enough when a guaranteed low worst-case cost per operation is required. (E.g., for real-time or parallel systems.)
- We use dynamic arrays as a simple application to illustrate amortized analysis.
- Recall that inserting a new element at the end is the only costly operation; all other operations (are assumed to) run in $O(1)$ time in the worst case.
- Hence, we can focus on sequences of insertions.
- Three approaches to amortized analysis:
  - Aggregate analysis;
  - Accounting method;
  - Potential method.
- We apply the first two methods to the analysis of dynamic arrays.
- Note, though, that these three methods need not be equally suited for the analysis of some particular problem.
Amortized Analysis: Aggregate Method for Analyzing Dynamic Arrays

- Aggregate analysis determines an upper bound $U(n)$ on the total cost of a sequence of $n$ operations.
- Then the amortized cost per operation is $U(n)/n$, i.e., all types of operations performed in the sequence have the same amortized cost.
- Suppose that the initial capacity of the array is 1.
- Then the cost $c_i$ of the $i$-th insertion is
  
  \[ c_i = \begin{cases} 
  i & \text{if } i - 1 \text{ is a power of } 2, \\
  1 & \text{otherwise.} 
  \end{cases} \]

- We get for the cost of $n$ insertions

  \[
  U(n) = \sum_{i=1}^{n} c_i = \left( \sum_{i=1}^{n} 1 \right) + \left( \sum_{i=1}^{n} ((i - 1) + 1) \right) \\
  = \sum_{i=0}^{\lfloor \log n \rfloor} 2^i = n + 2^{\lfloor \log n \rfloor + 1} - 1 \leq n + 2 \cdot 2^{\log n} = n + 2n = 3n.
  \]

- Hence, the amortized cost per (insertion) operation is $U(n)/n = \frac{3n}{n} \in O(1)$. 
Amortized Analysis: Accounting Method

- One of the main shortcomings of aggregate analysis is that different types of operations are assigned the same amortized cost.
- As a natural improvement, one might want to assign different costs to different operations.
- The accounting method (aka banker’s method) assigns charges to each type of operation.
- The amount charged for each type of operation is the amortized cost for that type.
- Some operations are overcharged while some other operations are undercharged.
- The balance is kept in a bank account:
  - Overcharged operations: If the charge is greater than the actual cost, then money can be saved and a *credit* accumulates in the bank account.
  - Undercharged operations: If the charge is less than the actual cost, then money is taken from the bank count to compensate the excess cost.
Amortized Analysis: Accounting Method

No debt!

Denote the (real) cost of the \( i \)-th operation by \( c_i \) and the amortized cost (i.e., charge) by \( \hat{c}_i \). Then we require

\[
\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i
\]

for every \( n \in \mathbb{N} \) and every sequence of \( n \) operations.

- If the charging scheme is not entirely trivial then one will have to resort to induction, loop invariants (or the like) in order to prove that the charging scheme of the accounting method works.
Recall that the cost $c_i$ of the $i$-th insertion at the end of the dynamic array is

$$c_i = \begin{cases} i & \text{if } i - 1 \text{ is a power of 2,} \\ 1 & \text{otherwise,} \end{cases}$$

while all other operations have a cost of 1.

We set the charge $\hat{c}_i$ for the $i$-th operation to 3 if it is an insertion, and to 1 otherwise.

We claim that this charging scheme will result in a bank account that is always positive.

Since all operations except insertions cost as much as we pay for, insertions are the only operations that we need to care about.
Amortized Analysis: Accounting Method

Charging scheme keeps positive account

The bank account is always positive after the insertion of the \( i \)-th element, for all \( i \in \mathbb{N} \).

Proof:
I.B.: For \( i = 1 \) there is one element in the array and \( \hat{c}_1 - c_1 = 2 \) in the bank account. For \( i = 2 \) we have two elements in the array and \( 2 + 3 - 2 = 3 \) in the bank account.
I.H.: The bank account is positive after the insertion of the \( j \)-th element, for some arbitrary but fixed \( i \in \mathbb{N} \) with \( i \geq 2 \) and all \( j \in \{1, 2, \ldots, i\} \).
I.S.: Since \( \hat{c}_{i+1} - c_{i+1} = 2 \), the bank account remains positive if the insertion of the \((i + 1)\)-st element does not require to resize the array.
So suppose that \((i + 1) - 1 = 2^k\) for \( k \in \mathbb{N} \). By the I.H. we know that the bank account was not negative when we doubled from a capacity of \( 2^{k-1} \) to \( 2^k \). After doubling we inserted \( 2^{k-1} \) new elements into the table of capacity \( 2^k \), saving \( 2 \cdot 2^{k-1} = 2^k \). This credit can be used to move all \( 2^k \) elements when doubling from \( 2^k \) to \( 2^{k+1} \), and the bank account contains at least \( 3 - 1 = 2 \) after the insertion of the element with number \( i + 1 = 2^k + 1 \).
Amortized Analysis of Increments of a Binary Counter

- If we need to store a (possibly large) binary counter then it is natural to resort to an array and let the array element $A[i]$ store the $i$-th bit of the counter.

- The standard way of incrementing the counter is to toggle the lowest-order bit. If that bit switches to a 0 then we toggle the next higher-order bit, and so forth until the bit that we toggle switches to a 1 at which point we can stop.

- If we have $n$ increment operations on a $k$-bit counter then the overall complexity is at most $O(k \cdot n)$. Note that, possibly, $k \gg n$. Can we do better?

- The result of the $i$-th increment is the number $i$ (if we started at 0). Hence, after $n$ increments at most $O(\log n)$ bits can have been toggled per increment, yielding a total of $O(n \log n)$ bits that need to be toggled.

- Is $O(n \log n)$ a tight bound? Can we do even better?

```c
Increment(binaryArray A) {
    i = 1;
    while ((i < A.length) && (A[i] != 0)) {
        A[i] = 0;
        ++i;
    }
    if (i < A.length) A[i] = 1;
}
```
Amortized Analysis of Increments of a Binary Counter

- **Aggregate method:** When does the $i$-th bit need to be toggled?
- In general, bit $A[i]$ is toggled $\lfloor n/2^{i-1} \rfloor$ many times.
- Hence, for a sequence of $n$ increments we get

$$\sum_{i=1}^{n} \left\lfloor \frac{n}{2^{i-1}} \right\rfloor = \sum_{i=0}^{n-1} \left\lfloor \frac{n}{2^i} \right\rfloor \leq \sum_{i=0}^{n-1} \frac{n}{2^i} = n \sum_{i=0}^{n-1} \frac{1}{2^i} \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

as the total amortized cost.
- Thus, we get 2 as the amortized cost of one increment.
Practical Relevance of Log-Terms

- Since $2^{20} = 1\,048\,576$ and $2^{30} = 1\,073\,741\,824$, in most applications the value of $\log n$ will hardly be significantly greater than 30 for practically relevant input sizes $n$.

- Hence, shaving off a multiplicative log-factor might constitute an important accomplishment when seen from a purely theoretical point of view, but its practical impact is likely to be much more questionable.

- In particular, multiplicative constants hidden in the $O$-terms may easily diminish the actual difference in speed between, say, an $O(n)$-algorithm and an $O(n \log n)$-algorithm.

Run-time experiments

Do not rely purely on experimental analysis to “detect” a log-factor: The difference between $\log(1\,024) = \log 2^{10}$ and $\log(1\,073\,741\,824) = \log 2^{30}$ is just a multiplicative factor of three!
Impact of Compile-Time Optimization

- Optimizing compilers try to minimize important characteristics of a program, such as its CPU-time consumption.
- Some problems related to code optimization are $\mathcal{NP}$-complete or even undecidable.
- Hence, several heuristics are employed that transform a program to a (hopefully) semantically equivalent program.
- E.g., an optimizing compiler will attempt to keep frequently used variables in registers rather than in main memory.
- Optimization may also involve the re-ordering of code or loop unrolling.
- C/C++: `gcc -O2`.

No speed-up guaranteed

In general, an optimized code will run faster. But optimization is not guaranteed to improve performance in all cases! It may even impede performance . . .
Impact of Compile-Time Optimization: Matrix Multiplication

**Definition 61 (Matrix multiplication)**

Let $A$ be a matrix of size $m \times n$ and $B$ be a matrix of size $n \times p$; that is, the number of columns of $A$ equals the number of rows of $B$. Then $A \cdot B$ is the $m \times p$ matrix $C = [c_{ij}]$ whose $(i, j)$-th element is defined by the formula

$$c_{ij} := \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1} b_{1j} + \cdots + a_{in} b_{nj}.$$ 

- Standard way to code matrix multiplication (for square matrices):

```c
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        sum = 0;
        for (k = 0; k < n; k++) sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```
Impact of Compile-Time Optimization: Matrix Multiplication

- Sample timings (in milliseconds) for the multiplication of two square matrices (with random integer elements).
- Platform: Intel\textsuperscript{TM} Core\textsuperscript{TM} i7-6700 CPU @3.40 GHz.

<table>
<thead>
<tr>
<th>n</th>
<th>100</th>
<th>500</th>
<th>1000</th>
<th>2000</th>
<th>5000</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc -O0</td>
<td>2.27</td>
<td>257.68</td>
<td>1943.27</td>
<td>10224.13</td>
<td>216154.34</td>
</tr>
<tr>
<td>gcc -O2</td>
<td>0.68</td>
<td>72.58</td>
<td>356.09</td>
<td>3606.34</td>
<td>85682.03</td>
</tr>
</tbody>
</table>

- Note that $\frac{3606}{356} \approx 2.16^3$ and $\frac{85682}{3606} \approx 2.98^3$.

```c
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        sum = 0;
        for (k = 0; k < n; k++) sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```
Floating-Point Arithmetic and Compilers

- Be warned that the standard 387 floating-point unit on x86 processors uses 80bit registers and operators, while standard “double” variables are stored in 64bit memory cells.
- Hence, rounding to a lower precision is necessary whenever a floating-point variable is transferred from register to memory.
- Optimizing compilers analyze code and keep variables within the registers whenever this makes sense, without storing intermediate results in memory.
- As a consequence of this excess precision of the register variables, on my PC,

\[ \sum_{i=1}^{10000000} 0.001 = 1000.000000000009095 \quad \text{with } gcc \ -O2 \ -mfpmath=387, \]

\[ \sum_{i=1}^{10000000} 0.001 = 999.999999832650701 \quad \text{with } gcc \ -O0 \ -mfpmath=387. \]

- Newer chips also support the SSE instruction set, and the default option -mfpmath=sse avoids this particular problem for x86-64 compilers.

**Warning**
The result of fp-computations may depend on the compile-time options!
Dealing with Floating-Point Computations

- Theory tells us that we can approximate the first derivative $f'$ of a function $f$ at the point $x_0$ by evaluating $\frac{f(x_0+h)-f(x_0)}{h}$ for sufficiently small values of $h$.
- Consider $f(x) := x^3$ and $x_0 := 10$:
  
  \[
  \begin{array}{lll}
  h := 10^0 : & f'(10) \approx 331.0000000 & h := 10^{-1} : & f'(10) \approx 303.0099999 \\
  h := 10^{-2} : & f'(10) \approx 300.3000999 & h := 10^{-3} : & f'(10) \approx 300.0300009 \\
  h := 10^{-4} : & f'(10) \approx 300.0030000 & h := 10^{-5} : & f'(10) \approx 300.0002999 \\
  h := 10^{-6} : & f'(10) \approx 300.0000298 & h := 10^{-7} : & f'(10) \approx 300.0000003 \\
  h := 10^{-8} : & f'(10) \approx 300.0000219 & h := 10^{-9} : & f'(10) \approx 300.0000106 \\
  h := 10^{-10} : & f'(10) \approx 300.0002379 & h := 10^{-11} : & f'(10) \approx 299.9854586 \\
  h := 10^{-12} : & f'(10) \approx 300.1332515 & h := 10^{-13} : & f'(10) \approx 298.9963832 \\
  h := 10^{-14} : & f'(10) \approx 318.3231456 & h := 10^{-15} : & f'(10) \approx 568.4341886 \\
  h := 10^{-16} : & f'(10) \approx 0.000000000 & h := 10^{-17} : & f'(10) \approx 0.00000000 \\
  \end{array}
  \]

- The cancellation error increases as the step size, $h$, decreases. On the other hand, the truncation error decreases as $h$ decreases.
- These two opposing effects result in a minimum error (and “best” step size $h$) that is high above the machine precision!
Dealing with Floating-Point Computations

- This gap between the theory of the reals and floating-point practice has important and severe consequences for the actual coding practice when implementing (geometric) algorithms that require floating-point arithmetic:
  1. The correctness proof of the mathematical algorithm does not extend to the program, and the program can fail on seemingly appropriate input data.
  2. Local consistency need not imply global consistency.

Numerical analysis . . .

. . . and adequate coding are a must when implementing algorithms that deal with real numbers. Otherwise, the implementation of an algorithm may turn out to be absolutely useless in practice, even if the algorithm (and even its implementation) would come with a rigorous mathematical proof of correctness!
Impact of Cache Misses

- Today’s computers perform arithmetic and logical operations on data stored in registers.
- In addition to main memory, data can also be stored in a Level 1 cache or a Level 2 cache. (Multi-core machines tend to have also L3 caches.)
- In a nutshell:
  - A cache is a fast but expensive memory which holds the values of standard memory locations.
  - If the CPU requests the value of a memory location and if that value is available in some level of the cache, then the value is fetched from the cache, at a cost of a few cycles: *cache hit*.
  - Otherwise, a block of consecutive memory locations is accessed and brought into the cache: *cache miss*.
  - A cache miss is much costlier than a cache hit!
- Since the gap between CPU speed and memory speed gets wider and wider, good cache management and programs that exhibit good *locality* become increasingly more important.
Impact of Cache Misses: Matrix Multiplication Revisited

- C/C++: Elements within the same row of a matrix are stored in consecutive memory locations, while elements in the same column may be far apart in main memory.
- The standard implementation of matrix multiplication causes the elements of $A$ and $C$ to be accessed row-wise, while the elements of $B$ are accessed by column.
- This will result in a lot of cache misses if $B$ is too large to fit into the (L2) cache.

```c
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        sum = 0;
        for (k = 0; k < n; k++) sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }
}
```
Impact of Cache Misses: Matrix Multiplication Revisited

- Rewriting of the standard multiplication algorithm ("ijk-order").
- Re-ordering of the inner loops will cause the matrices $B$ and $C$ to be accessed row-wise within the inner-most loop, while the indices $i, k$ of the $(i, k)$-th element of $A$ remain constant: "ikj-order".

```plaintext
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) c[i][j] = 0;
}
for (i = 0; i < n; i++) {
    for (k = 0; k < n; k++) {
        for (j = 0; j < n; j++) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```
Impact of Cache Misses: Matrix Multiplication Revisited

- Platform: Intel™ Core™ i7-6700 CPU @3.40 GHz.
- Caches: 256KiB L1, 1MiB L2, 8MiB L3.
- CPU-time consumption of ikj-order matrix multiplication divided by the CPU-time consumption of the standard ijk-order matrix multiplication.

<table>
<thead>
<tr>
<th></th>
<th>100</th>
<th>500</th>
<th>1000</th>
<th>2000</th>
<th>5000</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc -00 ikj/ijk</td>
<td>1.596</td>
<td>1.112</td>
<td>1.090</td>
<td>0.911</td>
<td>0.678</td>
</tr>
<tr>
<td>gcc -02 ikj/ijk</td>
<td>0.882</td>
<td>0.805</td>
<td>0.719</td>
<td>0.602</td>
<td>0.389</td>
</tr>
</tbody>
</table>

**Cache misses**

Avoiding cache misses may result in a substantially faster program!

**Algorithm engineering . . .**

. . . should be standard when designing and implementing an algorithm! Decent algorithm engineering may pay off more significantly than attempting to implement a highly complicated algorithm just because its theoretical analysis predicts a better running time.
Algorithmic Paradigms
- Incremental Construction
- Greedy
- Divide and Conquer
- Dynamic Programming
- Randomization
- Sweep
Incremental Construction

**Incremental construction**

A result $R(\{x_1, x_2, \ldots, x_n\})$ that depends on $n$ input items $x_1, x_2, \ldots, x_n$ is computed by dealing with one item at a time: For $2 \leq i \leq n$, we obtain $R(\{x_1, x_2, \ldots, x_i\})$ from $R(\{x_1, x_2, \ldots, x_{i-1}\})$ by “inserting” the $i$-th item $x_i$ into $R(\{x_1, x_2, \ldots, x_{i-1}\})$.

**Important invariant of incremental construction**

$R(\{x_1, x_2, \ldots, x_i\})$ exhibits all the desired properties of the final result $R(\{x_1, x_2, \ldots, x_n\})$ restricted to $\{x_1, x_2, \ldots, x_i\}$ as input items: If we would stop incremental construction after having inserted $x_i$ then we would have the correct solution $R(\{x_1, x_2, \ldots, x_i\})$ for $\{x_1, x_2, \ldots, x_i\}$.

- Once this invariant has been established the overall correctness of an incremental algorithm is a simple consequence.
- The total complexity is given as a sum of the complexities of the individual “insertions”.
- Incremental algorithms are particularly well suited for dealing with “online problems”, for which data items arrive one after the other. (Of course, only if you can afford the time taken by the subsequent insertions.)
Incremental Construction: Insertion Sort

- Insertion sort is a classical incremental algorithm: We insert the $i$-th item (of $n$ items) into the sorted list of the first $i-1$ items, thereby transforming it into a sorted list of the first $i$ items.

```
InsertionSort(array A[], int low, int high)
{
    for (i = low+1; i <= high; ++i) {
        x = A[i];
        j = i;
        while ((j > 1) && (A[j-1] > x)) {
            A[j] = A[j - 1];
            --j;
        }
        A[j] = x;
    }
}
```

- $O(n)$ for pre-sorted input, $O(n^2)$ worst case; efficient for small arrays.
- Adaptive (i.e., efficient for substantially sorted input), stable, in-place and online.
- Library sort maintains small chunks of unused spaces throughout the array and runs in $O(n \log n)$ time with high probability [Farach-Colton&Mosteiro 2006].
Incremental Construction: Convex Hull

Problem: CONVEXHULL

Input: A set $S$ of $n$ points in the Euclidean plane $\mathbb{R}^2$.
Output: The convex hull $CH(S)$, i.e., the smallest convex super set of $S$.

Lemma 62

1. The convex hull of a set $S$ of points in $\mathbb{R}^2$ is a convex polygon.
2. Two distinct points $p, q \in S$ define an edge of $CH(S)$ if and only if all points of $S \setminus \{p, q\}$ lie on one side of the line through $p, q$. 
Incremental Construction: Convex Hull

1. Sort the points according to their $x$-coordinates, and re-number accordingly:
   \[ S := \{p_1, p_2, \ldots, p_n\} \]. (Suppose that all $x$-coordinates are distinct.)

2. Compute \( CH(\{p_1, p_2, p_3\}) \).

3. Suppose that \( CH(\{p_1, p_2, \ldots, p_{i-1}\}) \) is known. We insert \( p_i \).
   a. Compute the supporting lines of \( CH(\{p_1, p_2, \ldots, p_{i-1}\}) \) and \( p_i \).
   b. Split \( CH(\{p_1, p_2, \ldots, p_{i-1}\}) \) into two parts at the two vertices of \( CH(\{p_1, p_2, \ldots, p_{i-1}\}) \) where the supporting lines touch.
   c. Discard that part of \( CH(\{p_1, p_2, \ldots, p_{i-1}\}) \) which faces \( p_i \).

What is the complexity of this incremental construction scheme?

Corollary 56: The worst-case complexity of \textsc{ConvexHull} for \( n \) points has an \( \Omega(n \log n) \) lower bound in the ACT model.
Incremental Construction: Convex Hull

Naïve complexity analysis:
- The initial sorting takes $O(n \log n)$ time.
- The construction of $CH(\{p_1, p_2, p_3\})$ takes $O(1)$ time.
- Inserting $p_i$ into $CH(\{p_1, p_2, \ldots, p_{i-1}\})$ will result in discarding one or more vertices of $CH(\{p_1, p_2, \ldots, p_{i-1}\})$ and, thus, takes $O(i)$ time.
- Hence we get
  $$O(n \log n) + O(1) + O(4 + 5 + \ldots + n) = O(n^2)$$
as total time complexity.

Amortized complexity analysis:
- Let $m_i$ denote the number of vertices that are discarded from $CH(\{p_1, p_2, \ldots, p_{i-1}\})$ when $p_i$ is inserted.
- Then the insertion of $p_i$ takes $O(m_i + 1)$ time.
- Observation: $m_4 + m_5 + \ldots + m_n < n$.
- Hence, the insertion of $p_i$ runs in amortized time $O(1)$, and the total complexity of the incremental construction algorithm is $O(n \log n)$.

Theorem 63
The convex hull of $n$ points in the plane can be computed in worst-case optimal time $O(n \log n)$ by means of incremental construction.
Greedy Paradigm

- A greedy algorithm attempts to solve an optimization problem by repeatedly making the locally best choice in a hope to arrive at the global optimum.

- A greedy algorithm may but need not end up in the optimum. E.g., greedy solution for ETSP-COMP or cashier's algorithm for coin changing.

- If a greedy algorithm is applicable to a problem then the problem tends to exhibit the following two properties:
  
  **Optimal substructure:** The optimum solution to a problem consists of optimum solutions to its sub-problems.
  
  **Greedy choice property:** Choices may depend on prior choices, but must not depend on future choices; no choice is reconsidered.

- If a greedy algorithm does indeed produce the optimum solution then it likely is the algorithm of choice since it tends to be faster than other algorithms (e.g., based on dynamic programming).

- Success stories: Kruskal’s algorithm and Prim’s algorithm for computing minimum spanning trees.
Greedy Paradigm: Selection Sort

- Selection sort is a classical greedy algorithm: We sort the array by repeatedly searching the $k$-smallest item and moving it forward to make it the $k$-th item of the array.

```c
SelectionSort(array A[], int low, int high) {
    for (i = low; i <= high; ++i) {
        int j_min = i;
        for (j = low+1; j <= high; ++j) {
            if (A[j] < A[j_min]) j_min = j;
        }
        if (j_min != i) Swap(A[i], A[j_min]);
    }
}
```

- Selection sort runs in $O(n^2)$ time in both the average and the worst case. Its running time tends to be inferior to that of insertion sort.
- However, it requires only $O(n)$ write operations of array elements while insertion sort may consume $O(n^2)$ write operations.
Greedy Paradigm: Huffman Coding

- Standard encoding schemes of symbols use a fixed number of bits to encode each symbol.
- E.g., ASCII encoding uses seven bits to encode the lowest 128 Unicode characters, from U+0000 to U+007F.
- Decoding an ASCII string is easy: Scan it in chunks of seven bits and look up the corresponding symbol.
- Encodings like ASCII do not take the frequency of occurrence of the individual symbols into account.
- One can achieve a (lossless) compression by encoding symbols that occur
  - more frequently (such as the letters “e” and “a”) with shorter bit strings;
  - less frequently (such as the letter “q”) with longer bit strings.
- Obvious problem for variable-length encodings: If one would assign, say, 1 to “a” and 11 to “q” then an encoding string that starts with 11 cannot be decoded unambiguously.
Greedy Paradigm: Huffman Coding

**Definition 64 (Prefix code, Dt.: Präfixcode, präfixfreier Code)**

Consider a set $\Omega$ of symbols. A *prefix code* for $\Omega$ is a function $c$ that maps every $x \in \Omega$ to a binary string, i.e., to a sequence of 0s and 1s, such that $c(x)$ is not a prefix of $c(y)$ for all $x, y \in \Omega$ with $x \neq y$.

$$\Omega := \{a, d, s\}$$

$c(a) := 0$

$c(d) := 10$

$c(s) := 11$

- A prefix code is also said to have the *prefix property*.
- Real-world examples of prefix codes:
  - Country dial-in codes used by member countries of the International Telecommunication Union.
  - Machine language instructions of most computer architectures.
  - Country and publisher encoding within ISBNs.

Hence, $001011$ corresponds to “aads”.

![Huffman Coding Diagram]
Greedy Paradigm: Huffman Coding

Lemma 65
Let $T$ be the binary tree that represents $c$. If $c$ is a prefix code for $\Omega$ then only the leaves of $T$ represent symbols of $\Omega$.

Definition 66 (Average number of bits per symbol)
Consider a set $\Omega$ of symbols and a frequency function $f : \Omega \to \mathbb{R}^+$. The average number of bits per symbol of a prefix code $c$ is given by

$$ANBS(\Omega, c, f) := \sum_{\omega \in \Omega} f(\omega) \cdot |c(\omega)|,$$

where $|c(\omega)|$ denotes the number of bits used by $c$ to encode $\omega$.

$c(a) := 0$ \hspace{1cm} $f(a) := 0.5$
$c(d) := 10$ \hspace{1cm} $f(d) := 0.3$
$c(s) := 11$ \hspace{1cm} $f(s) := 0.2$

Then, for $\Omega := \{a, d, s\}$, $ANBS(\Omega, c, f) = 0.5 \cdot 1 + 0.3 \cdot 2 + 0.2 \cdot 2 = 1.5$.

Definition 67 (Optimum prefix code)
A prefix code $c^*$ for $\Omega$ is optimum if it minimizes $ANBS(\Omega, c, f)$ for a given frequency $f$. 

Greedy Paradigm: Huffman Coding

- Recall that a binary tree $T$ is full if every non-leaf node of $T$ has two children.

**Lemma 68**
If a prefix code $c^*$ is optimum then the binary tree that represents $c^*$ is a full tree.

**Lemma 69**
The lowest-frequency symbol of $\Omega$ appears at the lowest level of the tree that represents an optimum prefix code $c^*$.

**Huffman’s greedy template (1952)**

1. Create two leaves for the two lowest-frequency symbols $s_1, s_2 \in \Omega$.
2. Recursively build the encoding tree for $(\Omega \cup \{s_{12}\}) \setminus \{s_1, s_2\}$, with $f(s_{12}) := f(s_1) + f(s_2)$, where $s_{12}$ is a new symbol that does not occur in $\Omega$.

**Theorem 70**
Huffman’s greedy algorithm computes an optimum prefix code $c^*$ for $\Omega$ relative to a given frequency $f$ of the symbols of $\Omega$. 
Greedy Paradigm: Job Scheduling

Problem: JOB SCHEDULING

**Input:** A set $J$ of $n$ jobs, where job $i$ starts at time $s_i$ and finishes at time $f_i$. Two jobs $i$ and $j$ are *compatible* if they do not overlap time-wise, i.e., if either $f_i \leq s_j$ or $f_j \leq s_i$.

**Output:** A maximum subset $J'$ of $J$ such that the jobs of $J'$ are mutually compatible.

Can we arrange the jobs in some “natural order”, and pick jobs successively provided that a new job is compatible with the previously picked jobs?
Greedy Paradigm: Job Scheduling

- Can we consider the jobs in some “natural order”?
  - Fewest conflicts: Pick jobs according to smallest number of incompatible jobs.
  - Shortest job duration: Pick jobs according to ascending order of $f_i - s_i$.
  - Earliest start time: Pick jobs according to ascending order of $s_i$.
  - Earliest finish time: Pick jobs according to ascending order of $f_i$.
Greedy Paradigm: Job Scheduling

Lemma 71

Picking jobs according to earliest finish time allows to compute an optimum solution to JOB_SCHEDULING in \( O(n \log n) \) time for \( n \) jobs.

Proof:

- It is obvious that sorting the \( n \) jobs in ascending order of \( f_i \) allows to generate a solution in \( O(n \log n) \) time. W.l.o.g., \( f_i \neq f_j \) if \( i \neq j \).
- Suppose that an optimum solution has \( m \) jobs while a greedy approach picks \( k < m \) jobs \( i_1, i_2, \ldots, i_k \).
- Let \( x \) be the largest-possible number such that \( i_1 = j_1, i_2 = j_2, \ldots, i_x = j_x \), over all optimum solutions \( j_1, j_2, \ldots, j_m \). We have \( x < m \).
- A compatible job \( i_{x+1} \) exists that finishes earlier than job \( j_{x+1} \), i.e., \( f_{i_{x+1}} < f_{j_{x+1}} \).
- Replacing job \( j_{x+1} \) by job \( i_{x+1} \) in the optimum solution maintains optimality, but violates maximality of \( x \).

\[
\text{greedy: } i_1 \quad i_2 \quad \ldots \quad i_x \quad i_{x+1} \quad \ldots \quad i_k \\
\text{opt: } j_1 \quad j_2 \quad \ldots \quad j_x \quad i_{x+1} \quad \ldots \quad i_m
\]
Greedy Paradigm: Processor Scheduling

Caveat
Seemingly similar problems may require different greedy strategies!

Problem: PROCESSORSCHEDULING

Input: A set $J$ of $n$ jobs, where job $i$ starts at time $s_i$ and finishes at time $f_i$. Two jobs $i$ and $j$ are compatible if they do not overlap time-wise.

Output: An assignment of all jobs to a minimum number of processors such that no two incompatible jobs run on the same processor.

Lemma 72
Assigning jobs according to earliest start time allows to compute an optimum solution to PROCESSORSCHEDULING in $O(n \log n)$ time.
Divide&Conquer: Merge Sort

Basic principle
Split the problem into sub-problems, recursively solve the sub-problems, and then merge the solutions of the subproblems to obtain the solution for the original problem.

```
1  MergeSort(array A[], int low, int high)
2  {
3      int i;        /* counter */
4      int middle;  /* index of middle element */
5      if (low < high) {
6          middle = (low+high) / 2;
7          MergeSort(A, low, middle);
8          MergeSort(A, middle+1, high);
9          Merge(A, low, middle, high);
10        }
11  }
```

- Always try to divide the job evenly!
- Does it matter if you cannot guarantee to split exactly in half? No! It is good enough to ensure that the size of every sub-problem is at most some constant fraction of the original problem size.
Divide&Conquer: Fast Matrix Multiplication

- Recall: If \( A, B \) are two square matrices of size \( n \times n \), then \( A \cdot B \) is the \( n \times n \) matrix \( C = [c_{ij}] \) whose \( (i, j) \)-th element \( c_{ij} \) is defined by the formula

\[
    c_{ij} := \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1} b_{1j} + \cdots + a_{in} b_{nj}.
\]

- Standard multiplication of two \( n \times n \) matrices results in \( \Theta(n^3) \) many arithmetic operations.

Theorem 73 (Strassen 1969)

Seven multiplications of scalars suffice to compute the multiplication of two \( 2 \times 2 \) matrices. In general, \( O(n^{\log_2 7}) \approx O(n^{2.807\ldots}) \) arithmetic operations suffice for \( n \times n \) matrices.

- Strassen’s algorithm is more complex and numerically less stable than the standard naïve algorithm. But it is considerably more efficient for large \( n \), i.e., roughly when \( n > 100 \), and it is very useful for large matrices over finite fields.
- It does not assume multiplication to be commutative and, thus, works over arbitrary rings.
Divide&Conquer: Fast Matrix Multiplication

Proof of Thm. 73 for \( n = 2 \): For \( A, B \in M_{2 \times 2} \), we compute

\[
C = A \cdot B = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}
\]

via

\[
\begin{align*}
p_1 & := (a_{12} - a_{22})(b_{21} + b_{22}) \\
p_2 & := (a_{11} + a_{22})(b_{11} + b_{22}) \\
p_3 & := (a_{11} - a_{21})(b_{11} + b_{12}) \\
p_4 & := (a_{11} + a_{12})b_{22} \\
p_5 & := a_{11}(b_{12} - b_{22}) \\
p_6 & := a_{22}(b_{21} - b_{11}) \\
p_7 & := (a_{21} + a_{22})b_{11}
\end{align*}
\]

and set

\[
\begin{align*}
c_{11} & := a_{11}b_{11} + a_{12}b_{21} = p_1 + p_2 - p_4 + p_6 \\
c_{12} & := a_{11}b_{12} + a_{12}b_{22} = p_4 + p_5 \\
c_{21} & := a_{21}b_{11} + a_{22}b_{21} = p_6 + p_7 \\
c_{22} & := a_{21}b_{12} + a_{22}b_{22} = p_2 - p_3 + p_5 - p_7.
\end{align*}
\]

Obviously, this approach results in 7 multiplications and 18 additions/subtractions.
Proof of Thm. 73 for $n = 2m$: For $A, B \in M_{2m \times 2m}$, we compute $C = A \cdot B$ by resorting to manipulating block matrices of size $m \times m$:

$\begin{pmatrix}
 a_{11} & \cdots & a_{1n} \\
 \vdots & \ddots & \vdots \\
 a_{n1} & \cdots & a_{nn}
\end{pmatrix}
\cdot
\begin{pmatrix}
 b_{11} & \cdots & b_{1n} \\
 \vdots & \ddots & \vdots \\
 b_{n1} & \cdots & b_{nn}
\end{pmatrix}
=
\begin{pmatrix}
 A_{11} & A_{12} \\
 A_{21} & A_{22}
\end{pmatrix}
\cdot
\begin{pmatrix}
 B_{11} & B_{12} \\
 B_{21} & B_{22}
\end{pmatrix},$

where $A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22} \in M_{m \times m}$ with

$A_{11} = \begin{pmatrix}
 a_{11} & \cdots & a_{1m} \\
 \vdots & \ddots & \vdots \\
 a_{m1} & \cdots & a_{mm}
\end{pmatrix},
A_{12} = \begin{pmatrix}
 a_{1,m+1} & \cdots & a_{1,2m} \\
 \vdots & \ddots & \vdots \\
 a_{m,m+1} & \cdots & a_{m,2m}
\end{pmatrix},$

$A_{21} = \begin{pmatrix}
 a_{m+1,1} & \cdots & a_{m+1,m} \\
 \vdots & \ddots & \vdots \\
 a_{2m,1} & \cdots & a_{2m,m}
\end{pmatrix},
A_{22} = \begin{pmatrix}
 a_{m+1,m+1} & \cdots & a_{m+1,2m} \\
 \vdots & \ddots & \vdots \\
 a_{2m,m+1} & \cdots & a_{2m,2m}
\end{pmatrix}.$

Analogously for $B_{11}, B_{12}, B_{21}, B_{22}$. Then the approach used for multiplying $2 \times 2$ matrices can be applied, with $a_{ij}$ and $b_{ij}$ being replaced by $A_{ij}$ and $B_{ij}$, for $1 \leq i, j \leq 2$. That is, we have matrices rather than scalars as operands for addition and multiplication.
Proof of Thm. 73 for \( n = 2m \) (cont’d): Hence, we can compute \( C = A \cdot B \) by using

- 7 multiplications of \( m \times m \) matrices,
- 18 additions/subtractions of \( m \times m \) matrices.

Obviously, one addition of two \( m \times m \) matrices takes \( O(m^2) \) time, i.e., \( O(n^2) \) time.

Let \( T(n) \) denote the number of (arithmetic) operations consumed by Strassen’s algorithm for multiplying two \( n \times n \) matrices. We get the following recurrence relation for \( T \):

\[
T(n) = 7 \cdot T \left( \frac{n}{2} \right) + O(n^2).
\]

The Master Theorem 31 yields

\[
T \in \Theta(n^\log_2^7) \approx O(n^{2.807\ldots}).
\]
Divide & Conquer: Fast Matrix Multiplication

- Strassen’s algorithm is not the fastest algorithm for multiplying matrices.

**Lemma 74 (Coppersmith & Winograd 1990)**

\[ O(n^{2.375\ldots}) \text{ arithmetic operations suffice for multiplying two } n \times n \text{ matrices.} \]

**Lemma 75 (Stothers 2010, 2013)**

\[ O(n^{2.37369}) \text{ arithmetic operations suffice for multiplying two } n \times n \text{ matrices.} \]

**Lemma 76 (Williams 2011, 2012, Le Gall 2014)**

\[ O(n^{2.37286\ldots}) \text{ arithmetic operations suffice for multiplying two } n \times n \text{ matrices.} \]

- Open problem: What is the true lower bound?

- The Coppersmith-Winograd algorithm and the more recent improvements are used frequently as building blocks in other algorithms to prove complexity bounds. E.g., the best algorithm for computing the diameter of a graph runs in \( O(n^\omega \log n) \) time, where \( \omega \) is the exponent of matrix multiplication.

- Besides Strassen’s algorithm, these algorithms are of no practical value, though, since the cross-over point for where they would improve on the naïve cubic-time algorithm is enormous.
Divide&Conquer: Closest Pair

Problem: CLOSESTPAIR

Input: A set \( S \) of \( n \) points in the Euclidean plane.

Output: Those two points of \( S \) whose mutual distance is minimum among all pairs of points of \( S \).

- Corollary 57: The worst-case complexity of CLOSESTPAIR for \( n \) points has an \( \Omega(n \log n) \) lower bound in the ACT model of computation.
- Easy to solve in \( O(n \log n) \) time if all points lie on the \( x \)-axis (or on a line).
Divide & Conquer: Closest Pair

**Problem: CLOSESTPAIR**

**Input:** A set $S$ of $n$ points in the Euclidean plane.

**Output:** Those two points of $S$ whose mutual distance is minimum among all pairs of points of $S$.

**Lemma 77**

$\text{CLOSESTPAIR}$ for $n$ points can be solved in worst-case optimal time $O(n \log n)$. 
Divide & Conquer: Closest Pair

Proof of Lemma 77:
- Sort the points according to $x$-coordinates, and split at median $x$-coordinate into a left sub-set and a right sub-set.
- Recursively find the minimum distance in the left sub-set and in the right sub-set. Return the points sorted according to $y$-coordinates.
- Consider a strip of width $2\delta$, where $\delta := \min\{\delta_l, \delta_r\}$.
- Slide a window of height $2\delta$ upwards within this strip and compute distances between those points of the left and the right sub-set which lie within this window.
- Merge the $y$-sorted points of the left and right sub-set.
**Proof of Lemma 77 (cont’d):** Time complexity:

- Sorting according to $x$-coordinates takes $O(n \log n)$ time.
- Splitting $n$ vertices at median $x$-coordinate takes $O(n)$ time.
- The distance computations take $O(1)$ time for each position of the sliding window.
- Thus, all distance computations carried out during the conquer step run in $O(n)$ time.
- Merging the $y$-sorted points of the left and right sub-set takes $O(n)$ time.
- Hence, for the time complexity $T(n)$ we get

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n),$$

resulting in $T \in O(n \log n)$,

and, thus, an overall $O(n \log n)$ time bound.

**Question:** How many points of the right sub-set can lie within the sliding window?

**Answer:** Only a constant number! (Eight is easy to argue, but one can prove six.)
Dynamic Programming

- In a nutshell, dynamic programming (DP) is a technique for efficiently implementing a recursive algorithm by storing results for sub-problems.
- It may be applicable if the naïve recursive algorithm would solve the same sub-problems over and over again. In that case, storing the solution for every sub-problem in a table to look up instead of re-compute may lead to a more efficient algorithm.
- The word “programming” in the term DP does not refer to classical programming at all. It was coined by Richard Bellman in 1957.
- According to John Rust [2006],

  *Bellman explained that he invented the name “dynamic programming” to hide the fact that he was doing mathematical research at RAND under a Secretary of Defense who ‘had a pathological fear and hatred of the term “research”.’ He settled on the term “dynamic programming” because it would be difficult to give a ‘pejorative meaning’ and because ‘It was something not even a Congressman could object to.’*

**Theorem 78 (Bellman-Held-Karp 1962)**

DP allows to solve TSP for \( n \) cities in \( O(n^2 2^n) \) time, at the cost of \( O(n 2^n) \) space.
Dynamic Programming: Fibonacci Numbers

- Recall that the Fibonacci numbers are defined as follows:
  
  \[
  F_n := \begin{cases} 
  n & \text{if } n \leq 1, \\
  F_{n-1} + F_{n-2} & \text{if } n \geq 2.
  \end{cases}
  \]

- Since the Fibonacci numbers are defined recursively, a recursive computation scheme seems natural . . .

- Note that this requires Fibonacci numbers to be computed over and over again.

- E.g., \( F_{n-4} \) is computed five times, each time from scratch.

- What is the complexity of this approach?
Dynamic Programming: Fibonacci Numbers

- If we ignore the cost of adding two (possibly large) integers then we get

\[ C(n) := C(n - 1) + C(n - 2) \]

as the cost \( C(n) \) for computing the \( n \)-th Fibonacci number.

- This is the same recurrence as for the Fibonacci numbers!

- The theory of Fibonacci numbers (Lem. 5) tells us that

\[ F_n = \frac{1}{\sqrt{5}} \cdot \left( \frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \cdot \left( \frac{1 - \sqrt{5}}{2} \right)^n. \]

- Hence, we pay an exponential price for computing \( F_n \) recursively:

\[ C \in O(\phi^n), \text{ with the golden ratio } \phi := \frac{1 + \sqrt{5}}{2}. \]

- The closed-form solution for \( F_n \) could be used to compute \( F_n \) using only \( \Theta(\log n) \) many multiplications.

- But this would require us to deal with irrational numbers!
A memoized DP approach allows to compute the $n$-th Fibonacci number in $O(n)$ steps (if we ignore the time needed to add large integers), using $O(n)$ memory.

A simple bottom-up DP approach suffices to compute the $n$-th Fibonacci number also in $O(n)$ steps but with $O(1)$ memory.

Note that it suffices to remember only the two numbers computed most recently.

```c
int Fibonacci(int number) /* greater zero */
{
    int n1 = 0;
    int n2 = 1;
    int temp, i;
    for (i = 1; i < number; ++i) {
        temp = n1 + n2;
        n1 = n2;
        n2 = temp;
    }
    return n2;
}
```
Dynamic Programming

Dynamic programming . . .

. . . may result in an efficient (sub-exponential) algorithm if the following conditions hold:

- A solution can be computed by combining solutions of sub-problems;
- A solution of every sub-problem can be computed by combining solutions of sub-subproblems; etc.
- Only a polynomial number of sub-problems occurs in total.

- **Memoization (top down):**
  - Apply standard recursion, but remember the solution to a previously solved sub-problem.
  - Re-use solution whenever a sub-problem is encountered that has already been solved.

- **Tabulation (bottom up):**
  - Build a table of solutions for the sub-problems in bottom-up fashion.

- **Complexity:** Roughly, we get the number of sub-problems times the complexity for solving every sub-problem.
Dynamic Programming: Matrix Chain Multiplication

- The standard method for multiplying a $p \times q$ matrix with a $q \times r$ matrix requires $p \cdot q \cdot r$ (scalar) multiplications and $p \cdot (q - 1) \cdot r$ additions, yielding a $p \times r$ result matrix.

- Recall that matrix multiplication is associative, but not commutative:
  \[(A \cdot B) \cdot C = A \cdot (B \cdot C)\] but, in general, \[A \cdot B \neq B \cdot A.\]

- Hence, if $A$ is a $100 \times 1$ matrix, $B$ is a $1 \times 100$ matrix, and $C$ is a $100 \times 1$ matrix, then
  
  - $(A \cdot B) \cdot C$ needs $(100 \cdot 1 \cdot 100) + (100 \cdot 100 \cdot 1) = 20000$ multiplications.
  - $A \cdot (B \cdot C)$ needs $(1 \cdot 100 \cdot 1) + (100 \cdot 1 \cdot 1) = 200$ multiplications!

- It is obvious that it may pay off to think about an optimal parenthesization.

**Problem: MatrixChainMultiplication**

**Input:** A sequence of $n$ matrices $A_1, A_2, \ldots, A_n$, where matrix $A_i$ has dimensions $d_{i-1} \times d_i$ for $i \in \{1, 2, \ldots, n\}$.

**Output:** An optimal parenthesization such that the standard computation of $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ results in the minimum number of multiplications.
Dynamic Programming: Matrix Chain Multiplication

- We can split the product of matrices into two products by multiplying the first $k$ matrices, multiplying the second $n - k$ matrices, and then multiplying the two resulting matrices:

$$A_1 \cdot A_2 \cdot \ldots \cdot A_n = (A_1 \cdot \ldots \cdot A_k) \cdot (A_{k+1} \cdot \ldots \cdot A_n)$$

Of course, $k$ can be any number out of $\{1, 2, \ldots, n - 1\}$.

**Optimality Observation**

If an optimal solution for $A_i \cdot \ldots \cdot A_j$ is given by $(A_i \cdot \ldots \cdot A_k) \cdot (A_{k+1} \cdot \ldots \cdot A_j)$, for $1 \leq i \leq k < j \leq n$, then also the parenthesizations of $A_i \cdot \ldots \cdot A_k$ and $A_{k+1} \cdot \ldots \cdot A_j$ need to be optimal.
Dynamic Programming: Matrix Chain Multiplication

- For $1 \leq i \leq j \leq n$, let $m[i, j]$ denote the minimum number of scalar multiplications needed to compute $A_i \cdot \ldots \cdot A_j$.
- Recall that $A_i$ has dimensions $d_{i-1} \times d_i$. Hence, for $i \leq k < j$,

  $$(A_i \cdot \ldots \cdot A_k) \cdot (A_{k+1} \cdot \ldots \cdot A_j)$$

  has dimensions $d_{i-1} \times d_j$.

- We get the following formula:

  $$m[i, j] = \begin{cases} 
  0 & \text{if } i = j, \\
  \min_{i \leq k < j} \{m[i, k] + m[k+1, j] + d_{i-1}d_kd_j\} & \text{if } i < j.
  \end{cases}$$

- As in the case of the Fibonacci numbers, a naïve evaluation of this recursive formula would result in re-computing partial solutions over and over again.
- We resort to dynamic programming, and tabulate $m[i, j]$ as it becomes known.
- In the pseudo code on the next slide, we use $s[i, j]$ to store the optimum value of $k$ for splitting $A_i \cdot \ldots \cdot A_j$ into $(A_i \cdot \ldots \cdot A_k) \cdot (A_{k+1} \cdot \ldots \cdot A_j)$.
- The dimensions of $A_1, A_2, \ldots, A_n$ are stored in the array $d[]$. 
Dynamic Programming: Matrix Chain Multiplication

```c
void MatrixChainMultiplication(int d[], int s[])
{
    int seq_len, cost;
    int N = d.length - 1;
    for (i = 1; i <= N; i++) m[i, i] = 0;
    for (seq_len = 2; seq_len <= N; ++seq_len) {
        for (i = 1; i <= N - seq_len + 1; ++i) {
            j = i + seq_len - 1;
            m[i, j] = MAXINT; // "infinity"
            for (k = i; k <= j - 1; ++k) {
                cost = m[i, k] + m[k+1, j] +
                       d[i-1] * d[k] * d[j];
                if (cost < m[i, j]) {
                    m[i, j] = cost; // minimum cost so far
                    s[i, j] = k; // index of best split
                }
            }
        }
    }
}
```

Dynamic Programming: Matrix Chain Multiplication

- It is an easy exercise to extract the actually best parenthesization from $s[]$:

```java
string GetParenthesization(int i, int j, int s[]) {
    if (i < j) {
        x = GetParenthesization(i, s[i,j], s);
        y = GetParenthesization(s[i,j] + 1, j, s);
        return "(x * y)";
    }
    else
        return "A_{" # IntToString(i);
}
```

- Hence we get the following result:

**Theorem 79**

MATRIXCHAINMULTIPLICATION can be solved in $O(n^3)$ time and $O(n^2)$ space for $n$ matrices.
Dynamic Programming: Minimum-Weight Triangulation

**Definition 80 (Simple polygon; Dt.: einfaches Polygon)**

A simple (closed) *polygon* with vertices \( v_1, v_2, \ldots, v_n \) is a sequence of \( n \) straight-line segments ("edges") \( v_1v_2, v_2v_3, \ldots, v_{n-1}v_n \) and \( v_nv_1 \) such that no two edges intersect except for subsequent edges sharing a common vertex.

**Definition 81 (Diagonal)**

A *diagonal* of a simple polygon \( P \) with vertices \( v_1, v_2, \ldots, v_n \) is a straight-line segment \( v_iv_j \) defined by two distinct and non-consecutive vertices \( v_i, v_j \) of \( P \) which lies completely within the area bounded by \( P \).
Dynamic Programming: Minimum-Weight Triangulation

**Definition 82 (Triangulation)**

A *triangulation* of a simple polygon $\mathcal{P}$ with $n$ vertices is a partitioning of the area bounded by $\mathcal{P}$ into $n - 2$ triangles by inserting $n - 3$ diagonals which do not intersect each other except for possibly sharing common vertices of $\mathcal{P}$.

**Problem: MINIMUM WEIGHT TRIANGULATION (MWT)**

**Input:** A simple polygon $\mathcal{P}$ with $n$ vertices.

**Output:** A triangulation $\mathcal{T}$ of $\mathcal{P}$ such that the sum of the edge lengths of the diagonals of $\mathcal{P}$ that induce $\mathcal{T}$ is minimum over all triangulations of $\mathcal{P}$. 
Dynamic Programming: Minimum-Weight Triangulation

- We simplify the problem and consider only convex polygons $P$.
- Obviously, every pair of non-consecutive vertices of $P$ defines a diagonal of $P$.

**Optimality Observation**

If we knew that a diagonal — say $v_2 v_5$ — occurs in a MWT of $P$, then we could obtain the full MWT by computing MWTs for the two sub-polygons induced by the diagonal. (That is, for $(v_2, v_3, v_4, v_5, v_2)$ and $(v_2, v_5, v_6, v_7, v_8, v_9, v_1, v_2)$.)

- Again, simply enumerating all diagonals and recursively finding the MWT would be too costly. (See Lemma 87 on the number of triangulations of $P$.)
Dynamic Programming: Minimum-Weight Triangulation

Lemma 83

Consider an \( n \)-vertex convex polygon \( \mathcal{P} \). Select one edge of \( \mathcal{P} \) and denote it by \( r \). There is a bijection between the triangulations of \( \mathcal{P} \) and full binary trees with \( n - 1 \) leaves if we match the edge \( r \) with the root of the tree.

**Sketch of Proof:** We number the edges in CCW order, starting at (but excluding) \( r \).

- Every triangulation of \( \mathcal{P} \) corresponds to a unique full binary tree with \( n - 1 \) leaves such that the root of the tree corresponds to \( r \). (Every triangle corresponds to one inner node.)
**Lemma 83**

Consider an $n$-vertex convex polygon $P$. Select one edge of $P$ and denote it by $r$. There is a bijection between the triangulations of $P$ and full binary trees with $n - 1$ leaves if we match the edge $r$ with the root of the tree.

**Sketch of Proof:** We number the edges in CCW order, starting at (but excluding) $r$.

- Every triangulation of $P$ corresponds to a unique full binary tree with $n - 1$ leaves such that the root of the tree corresponds to $r$. (Every triangle corresponds to one inner node.)
- Every full binary tree with $n - 1$ leaves corresponds to a unique triangulation of $P$ such that the root of the tree corresponds to the triangle attached to $r$. 

[Diagram of a convex polygon and a corresponding full binary tree]
Lemma 84

Consider an $n$-vertex convex polygon $P$. Select one edge of $P$ and denote it by $r$. Label the other edges of $P$ by $A_1, A_2, \ldots, A_{n-1}$ in CCW order, starting at $r$. Then there exists a bijection between the triangulations of $P$ and parenthesizations of the matrix chain product $A_1 \cdot A_2 \cdot \ldots \cdot A_{n-1}$.

**Sketch of Proof:** We label the edges in CCW order, starting at (but excluding) $r$.

- Every triangulation of $P$ corresponds to a unique parenthesization of $A_1 \cdot A_2 \cdot \ldots \cdot A_{n-1}$.
- Every parenthesization of $A_1 \cdot A_2 \cdot \ldots \cdot A_{n-1}$ corresponds to a unique triangulation of $P$. 

$$((A_1 \cdot (A_2 \cdot A_3)) \cdot A_4) \cdot (A_5 \cdot ((A_6 \cdot A_7) \cdot A_8))$$
Dynamic Programming: Minimum-Weight Triangulation

- No need to use the actual lengths as weights for the diagonals. Rather, we can use arbitrary (positive) weights.
- In particular, we can assign the matrix dimensions to the vertices, and obtain the cost of a triangle as the product of the dimensions associated with its vertices.
- The total cost is given by the sum of the costs of the triangles.
- Then a solution to that variant of MWT corresponds to a solution of the original instance of `MATRIX_CHAIN_MULTIPLICATION`.

The cost of the shaded triangle is given by $d_0 \cdot d_1 \cdot d_4$; it corresponds to the cost of multiplying $A_1$ with $(A_2 \cdot A_3 \cdot A_4)$. 
Dynamic Programming: Minimum-Weight Triangulation

**Theorem 85 (Hu&Shing 1982)**

MINIMUM WEIGHT TRIANGULATION can be solved in $O(n \log n)$ time and $O(n)$ space for a convex polygon with $n$ vertices.

**Corollary 86 (Hu&Shing 1982)**

MATRIX CHAIN MULTIPLICATION can be solved in $O(n \log n)$ time and $O(n)$ space for $n$ matrices.

- Note that the following greedy approach need not yield the optimum:
  1. Select the cheapest diagonal.
  2. Recurse within the two sub-polygons induced.
- However, this greedy approach gives a triangulation that is within a constant factor of the optimum for convex polygons.
- It is easy to extend the DP approach to general polygons while maintaining the $O(n^3)$ worst-case complexity.
- [Seidel 1991]: A simple $n$-vertex polygon can be decomposed into trapezoids and triangulated in expected $O(n \log^* n)$ time.
- [Chazelle 1991]: A simple $n$-vertex polygon can be decomposed into trapezoids and triangulated in $O(n)$ time.
Lemma 87

There are $C_m$ different full binary trees with $m + 1$ leaves.

Proof: For $m \in \mathbb{N}_0$, let $T_m$ denote the number of different full binary trees with $m$ internal nodes, i.e., with $m + 1$ leaves. Obviously, $T_0 = 1$.

Now consider a full binary tree with $m \geq 1$ internal nodes, and thus, $m + 1$ leaves. Such a tree consists of a left subtree with $k$ internal nodes and a right subtree with $m - 1 - k$ internal nodes, for some $k \in \{0, 1, \ldots, m - 1\}$. For any fixed particular $k \in \{0, 1, \ldots, m - 1\}$, we get $T_k$ different left subtrees and $T_{m - 1 - k}$ different right subtrees. Hence, $T_m$ fulfills the recurrence

$$T_m = \sum_{k=0}^{m-1} T_k \cdot T_{m-1-k}, \quad \text{with } T_0 = 1.$$

This is the recurrence for the Catalan numbers (Def. 6), and we get $T_m = C_m$. 

\[\]
Corollary 88
There are $C_{n-2}$ many different triangulations of convex polygon with $n$ vertices.

Proof: A bijection between triangulations of a convex $n$-gon and full binary trees with $n-1$ leaves was established in Lem. 83.

Corollary 89
There are $C_{n-1}$ many different parenthesizations of the matrix chain product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$.

Proof: A bijection between triangulations of a convex $n$-gon and parenthesizations of the matrix chain product $A_1 \cdot A_2 \cdot \ldots \cdot A_{n-1}$ was established in Lem. 84.

Corollary 90
The numbers of different triangulations of a convex polygon with $n$ vertices and of different full binary trees with $n$ leaves are exponential in $n$.

Proof: We have, by Lemma 7, $C_n \in \Theta \left( \frac{4^n}{n^{1.5}} \right)$. 

**Deterministic vs. Randomized Algorithm**

**Deterministic algorithm:**
- It will always produce the same output in repeated runs for a particular input.
- The underlying state machine will always pass through the same sequence of states for the same input.
- Differences in running time for the same input are due to system-dependent reasons.
- Randomization and probabilistic methods play a key role in modern algorithm theory: Randomized algorithms are often simpler to understand and implement, while being correct and efficient with high probability.

**Randomized algorithm:**
- It uses a random number at least once to make a (branching) decision.
- Repeated runs for the same input may result in different outputs or running times.
- Probability of generating incorrect output.
- Efficiency is guaranteed only with some probability.
Randomization: Random Numbers

- “Classical” approaches like the rolling of dice or the flipping of coins cannot be used by computers.
- One alternative is to measure some physical phenomenon that can be expected to be random. E.g., the seconds of the current wall-clock time can be expected to yield a random number between 0 and 59.
- Most Unix/Linux-like operating systems have `/dev/urandom`, which allows to access environmental noise collected from sources like device drivers.
- The second alternative is to use an algorithm to generate [sic!] random numbers: pseudorandom number generator (PRNG).
- E.g., `arc4random()` is available on BSD platforms, and also on GNU/Linux with `libbsd`. It is an easy-to-use option for most standard C/C++-applications. (And it is much better than LCG-based generators like `rand()`!)

Practical advice

- Invest more time into testing since achieving path coverage becomes trickier!
- Employ randomization in such a way that the algorithm’s behavior can be made reproducible — i.e., deterministic! — if required: Debugging might be needed!
The following code generates a pseudorandom integer within the set \(\{from, \ldots, to\}\).

```c
int RandomNumberRange(const int from, const int to)
{
    int rnd, range = to - from + 1;
    int maxSafeRange = maxRndNumber - (maxRndNumber % range);

    do {
        rnd = GetRandomNumber(); // e.g., use arc4random()
    } while (rnd >= maxSafeRange);

    return from + (rnd % range);
}
```

Note that solutions simply resorting to the modulo operator, \%, to restrict a random number to a range of numbers tend to produce skewed results.

The skew in the distribution is made worse if the random number is obtained from an LCG since LCGs tend to have poor entropy in the lower bits.
Randomization: Random Numbers in C++

```cpp
#include <random>

std::random_device rnd_dev;
std::mt19937 gen(rnd_dev());

int RandomNumberRange(const int from, const int to) {
    std::uniform_int_distribution<int> distr(from, to);
    return distr(gen);
}
```

- The C++11 class `std::random_device` is a uniformly-distributed integer random number generator that makes use of a non-deterministic source if it is available.
- We use it to seed a Mersenne Twister PRNG based on the Mersenne prime $2^{19937} - 1$, which has a period of $2^{19937} - 1$.
- Then this PRNG is used to generate a 32-bit random number, which is mapped to the set \{from, \ldots, to\} via a call to `std::uniform_int_distribution`.
- The STL also contains a 64-bit implementation of the Mersenne Twister PRNG: `std::mt19937_64`. 
Randomization: Monte Carlo vs. Las Vegas

Monte Carlo algorithm:
- Is always fast.
- Might fail to produce a correct output, with one-sided or two-sided errors.
- The probability of an incorrect output is bound based on an error analysis.
- Repetitions of Monte Carlo algorithms tend to drive down the failure probability exponentially.

Las Vegas algorithm:
- Always gives a correct output (or reports an error).
- Its run-time performance may vary.
- Several Las Vegas algorithms can be turned into Monte Carlo algorithms by setting a time budget and stopping the algorithm once this time budget is exceeded.
Random Permutation

Problem: RandomPermutation

Input: A sequence \( S = (s_0, s_1, \ldots, s_{n-1}) \) of \( n \) entities.

Output: A random permutation of these \( n \) entities, uniformly at random.

```c
void RandomPermutation(array S[]) // Knuth shuffle
{
    N = S.length - 1;
    for (i = N; i >= 1; --i) {
        j = RandomInteger({0,1,...,i});
        Swap(S[i], S[j]);
    }
}
```

- Knuth’s version of the Fisher-Yates shuffle [1938] runs in \( \Theta(n) \) time, with \( n := |S| \).
- After \texttt{RandomPermutation}(S) we have \( \Pr[s = s_i] = \frac{1}{n} \) for all \( s \in S \) and all \( i \in \{0, 1, \ldots, n - 1\} \).
- Hence \texttt{RandomPermutation}(S) generates each permutation with probability \( \frac{1}{n!} \), i.e., uniformly at random.
Randomized QuickSort

- We assume that all numbers of the array to be sorted are distinct.

**Standard QuickSort:**
1. Pick the left-most element \( p \) of the array as the pivot.
2. Rearrange and split the array into two subarrays LESS and GREATER by comparing each element to \( p \).
3. Recurse on LESS and GREATER.

- \( O(n^2) \) worst-case complexity, even when using median-of-three partitioning.
- One can specify worst-case input!
- \( O(n \log n) \) average-case complexity.

**Randomized QuickSort:**
1. Pick an element \( p \) of the array as the pivot, uniformly at random.
2. Rearrange and split the array into two subarrays LESS and GREATER by comparing each element to \( p \).
3. Recurse on LESS and GREATER.

- \( O(n \log n) \) expected-time complexity for all inputs of \( n \) numbers.
- One can also generate a random permutation of the input numbers and then run the standard QuickSort on that shuffled array.
Randomized QuickSort

**Theorem 91**

The expected number of comparisons made by a randomized QuickSort on an array of $n$ input numbers is at most $2n \ln n$.

**Proof:** We define the random variable $X_{ij}$ to be 1 if the algorithm does compare the $i$-th smallest element to the $j$-th smallest element, and 0 otherwise. Let $X$ denote the total number of comparisons. Since we never compare the same pair of elements twice, we get

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij},$$

and, due to the linearity of the expectation (Lemma 24),

$$\mathbb{E}(X) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}(X_{ij}).$$

Consider $X_{ij}$ for $1 \leq i < j \leq n$. We denote the $i$-th smallest element in the array by $e_i$ and the $j$-th smallest element by $e_j$. 
Randomized QuickSort

Proof of Theorem 91 (cont’d):

1. If we choose a pivot $p$ such that $e_i < p < e_j$ then $e_i$ ends up in LESS and $e_j$ ends up in GREATER, and $e_i$ and $e_j$ will never be compared.

2. If we choose $e_i$ or $e_j$ as pivot then we do compare them.

3. If we choose $p < e_i$ or $p > e_j$ then the decision is deferred, and we will pick a new pivot in the next recursive step.

At each step, the probability that $X_{ij} = 1$ under the condition that we will certainly not compare $e_i$ to $e_j$ in the future is exactly $\frac{2}{j-i+1}$. Hence, the overall probability of $X_{ij} = 1$ equals $\frac{2}{j-i+1}$, too.

Recall Lemma 9 on the Harmonic numbers $H_n$. We get

$$
\mathbb{E}(X) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}(X_{ij}) = \sum_{i=1}^{n-1} 2 \sum_{j=i+1}^{n} \frac{1}{j-i+1} = \sum_{i=1}^{n-1} 2 \left( \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-i+1} \right)
$$

$$
< 2n \left( \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \right) = 2n \cdot (H_n - 1) \leq 2n \ln n.
$$

Average case versus expected case

Since we average over all permutations (of some particular input!), this $O(n \log n)$ bound is a worst-case expected-time bound and applies even to (mostly) sorted input!
Randomized Primality Testing

**Problem: PRIME**

**Input:** A natural number \( n \) with \( n > 1 \).

**Decide:** Is \( n \) prime? I.e., can \( n \) be divided only by 1 and by itself?

- Note that we only care to know whether some number \( n \) is prime; we do not seek a prime factorization of \( n \).
- PRIME is a basic building block for many applications. E.g., large primes are frequently sought in cryptography. (A good public key uses prime numbers with at least 250 binary digits.)
- Brute-force methods — e.g., repeated divisions by 2, 3, \ldots, \( \lfloor \sqrt{n} \rfloor \) — are far too slow when \( n \) is truly large. (Exponential worst-case running time in the size of \( n \).)
- PRIME is solvable in polynomial time [Agrawal&Kayal&Saxena 2002]: Their algorithm runs in \( O(\log^{7.5+\varepsilon} n) \) time, which is polynomial in the size of \( n \). (In 2005, Pomerance&Lenstra reduced this to \( O(\log^{6+\varepsilon} n) \).
- But this is a rather theoretical result . . .
- Fortunately, large primes are not particularly rare.
- Expectation: One out of \( \ln n \) random integers of the size of \( n \) will be prime!
- Randomization yields an efficient, simple and easy-to-implement primality test — if we accept a small probability of error!
Randomized Primality Testing: Basic Idea

**Witness of compositeness**

Find a predicate $P$ and a suitable set $S$ (with, typically, $S \subseteq \mathbb{N}$) such that

$$p \in \mathbb{P} \implies (\forall s \in S \; P(s, p)).$$

This is equivalent to

$$(\exists s \in S \; \neg P(s, p)) \implies p \notin \mathbb{P}.$$  

- Suppose that $p \notin \mathbb{P}$ and that $\neg P(s, p)$ holds for, say, at least 50% of the elements $s \in S$.
- Then we incorrectly classify a number $p$ as prime
  - after testing $P(s, p)$ for just one $s \in S$ with a probability of at most $\frac{1}{2}$;
  - after testing $P(s, p)$ for just two $s \in S$ with a probability of at most $\frac{1}{4}$;
  - after testing $P(s, p)$ for $k$ numbers of $S$ with a probability of at most $\frac{1}{2^k}$. 
Randomized Primality Testing: Fermat

Lemma 92 (Fermat’s Little Theorem, Dt.: Kleiner Satz von Fermat)

If $p \in \mathbb{P}$ then $a^p \equiv_p a$ for every $a \in \mathbb{N}$.

- If $a$ is not a multiple of $p$ then this reduces to $a^{p-1} \equiv_p 1$.
- Hence, if $a^{n-1} \not\equiv_n 1$ for a given $n \in \mathbb{N}$ and some $a \in \{2, 3, \ldots, n-1\}$ then $n$ is composite, i.e., not a prime.
- Such an integer $a$ is called a Fermat witness for the compositeness of $n$.
- Otherwise, $n$ is possibly prime — or $a$ is a so-called Fermat liar.
- E.g., $2^{242} \mod 243 = 121$, implying that 243 is composite. And, indeed $243 = 3^5$.
- E.g., $2^{240} \mod 241 = 1$, implying that 241 could be prime. And, indeed, 241 is prime.
Randomized Primality Testing: Fermat

- However, $2^{560} \mod 561 = 1$, but $561 = 3 \cdot 11 \cdot 17$.
- Worse, $4^{560} \mod 561 = 1$ and $5^{560} \mod 561 = 1$ and $7^{560} \mod 561 = 1$ and $8^{560} \mod 561 = 1$ and …
- We have $a^{560} \mod 561 = 1$ for all $a \in \{2, 3, \ldots, 560\}$ for which $\gcd(561, a) = 1$. The number 561 is the smallest so-called \textit{Carmichael number}.
- Carmichael numbers are rather rare: There are about $2 \cdot 10^7$ Carmichael numbers between 1 and $10^{21}$, i.e., on average one Carmichael number within $5 \cdot 10^{13}$ numbers. (But there are infinitely many Carmichael numbers.)

Lemma 93
If $n$ is a composite number that is no Carmichael number then at least half of all $a \in \{2, 3, \ldots, n - 2\}$ are Fermat witnesses.

Theorem 94
If $n$ is a composite number that is no Carmichael number then $k$ rounds of the Fermat primality test (with $k$ randomly chosen values for $a \in \{2, 3, \ldots, n - 2\}$) will incorrectly classify $n$ as prime with probability at most $2^{-k}$. 
Randomized Primality Testing: Fermat

```cpp
bool IsPrimeFermat(int n, int k)
{
    A = {2,3,...,n-2};
    for (i = 1; i <= k; ++i) {
        a = RandomInteger(A);
        if (gcd(n, a) != 1) return false; /* composite */
        if ((a^(n-1) % n) != 1) return false; /* composite */
        A = A \ {a};
    }
    return true; /* probably prime */
}
```

- `IsPrimeFermat` is a Monte Carlo algorithm with one-sided error: it will classify all primes as “prime”, and falsely report a composite number as “prime” with probability at most $2^{-k}$. That is, it is correct with high probability.
- Note that the number $k$ of random trials need not be scaled with the size of $n$ in order to keep the error probability below $2^{-k}$.
- Still, the Fermat primality test is not considered to be reliable enough on its own grounds. It is, however, used for a rapid screening of possible candidate primes.
Randomized Primality Testing: Miller-Rabin

**Lemma 95**

Let $n \in \mathbb{N}$ be prime with $n > 2$, and $s, d \in \mathbb{N}_0$ such that $n - 1 = 2^s \cdot d$, with $d$ odd. Then for all $a \in \{2, 3, \ldots, n - 2\}$ we have

$$a^d \equiv_n 1 \quad \text{or} \quad a^{2^r \cdot d} \equiv_n -1 \quad \text{for some } r \in \{0, 1, \ldots, s - 1\}.$$  

The contrapositive of this lemma yields a test for compositeness:

**Lemma 96**

Let $n \in \mathbb{N}$ be odd with $n \geq 5$, and $s, d \in \mathbb{N}_0$ such that $n - 1 = 2^s \cdot d$, with $d$ odd. If there exists an $a \in \{2, 3, \ldots, n - 2\}$ such that

$$a^d \not\equiv_n 1 \quad \text{and} \quad a^{2^r \cdot d} \not\equiv_n -1 \quad \text{for all } r \in \{0, 1, \ldots, s - 1\},$$

then $n$ is composite.

Such an integer $a$ is called an **MR-witness** of compositeness.
Randomized Primality Testing: Miller-Rabin

```c
bool IsPrimeMillerRabin(int n, int k) /* for odd n > 2 */ {
    s = 0; d = n - 1;
    while (IsEven(d)) { /* (n-1) = 2^s*d with odd d */
        ++s; d /= 2;
    }
    A = {2,3,...,n-2};
    LOOP: for (i = 1; i <= k; ++i) {
        a = RandomInteger(A); A = A \ {a};
        x = a^d % n;
        if ((x == 1) || (x == -1)) do next LOOP;
        for (j = 1; j < s; ++j) {
            x = x^2 % n;
            if (x == 1) return false; /* composite */
            if (x == -1) do next LOOP;
        }
        return false; /* composite */
    }
    return true; /* probably prime */
}
```
Randomized Primality Testing: Miller-Rabin

**Lemma 97**

Let \( n \in \mathbb{N} \) be an odd composite number with \( n \geq 5 \). Then the set \{2, 3, \ldots, n − 2\} contains at most \( \frac{n-3}{4} \) numbers \( a \) such that \( \gcd(n, a) = 1 \) but \( a \) is no MR-witness of the compositeness of \( n \).

- Trivially, if \( \gcd(n, a) > 1 \) then \( a \) is always a witness of the compositeness of \( n \).

**Theorem 98**

If \( n \) is an odd composite number then the Miller-Rabin primality test with \( k \) rounds (and \( k \) randomly chosen values for \( a \in \{2, 3, \ldots, n − 2\} \)) will incorrectly classify \( n \) as prime with probability at most \( 4^{-k} \).

- Hence, 10 rounds of the Miller-Rabin primality test give us a probability of error that is \( 4^{-10} \approx 10^{-6} \), and 20 rounds result in a probability of error that is roughly \( 10^{-12} \). After 30 rounds we are down to roughly \( 10^{-18} \) error probability.
- For comparison purposes: Quality hard disks have a probability of about \( 10^{-16} \) for an unrecoverable read error (URE).
- Note that this error bound does not depend on the size of \( n \) and that it holds also for Carmichael numbers!
Randomized Primality Testing: Miller-Rabin

Lemma 99

One round of the Miller-Rabin primality test for input number $n$ takes $O(\log^3 n)$ time when using modular exponentiation by repeated squaring.

- FFT-based multiplication can bring the time complexity of one round down to $O(\log^2 n \cdot \log(\log n) \cdot \log(\log(\log n)))$.

- If the Generalized (aka Extended) Riemann Hypothesis (GRH) — which is a number-theoretic conjecture that is generally believed to be true — holds then for every composite number $n$ the set $\{1, 2, \ldots, \lfloor 2 \ln^2 n \rfloor \}$ contains an MR-witness for $n$. Hence, if one assumes the Extended Riemann Hypothesis then there is a deterministic algorithm to test primality in time $O(\log^5 n)$.

- [Jiang&Deng 2014]: If $n$ is “small” then smaller sets of potential MR-witnesses are known, with no need to resort to the GRH:
  - If $n < 2^{11} - 1 = 2047$: It suffices to test $a \in \{2\}$.
  - If $n < 2^{64}$: It suffices to test $a \in \{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37\}$.

- [Sorenson&Webster 2015] go even beyond 64-bit results:
  - If $n < 10^{24}$: It suffices to test $a \in \{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41\}$.
Line Segment Intersection

Problem: LINE_SEGMENT_INTERSECTION

Input: A set $S$ of line segments in $\mathbb{R}^2$.

Decide: Do any two segments of $S$ intersect?

- Two line segments are said to intersect if they share a point other than a common end-point.

- LINE_SEGMENT_INTERSECTION does not require us to find and report one or all intersections.

- Still, we explain how all intersections can be found.

- Stopping the algorithm at the first intersection (if one exists) yields an answer to the original problem.
The complexity of LINESEGMENTINTERSECTION for $n$ line segments has an $\Omega(n \log n)$ lower bound in the ACT model.

Proof:

- We use a reduction from ELEMENTUNIQUENESS. Let $\{x_1, x_2, \ldots, x_n\}$ be an instance of ELEMENTUNIQUENESS.
- We map $x_i$ to the line segment between the two points $(x_i, 0)$ and $(x_i, 1)$.
- Then ELEMENTUNIQUENESS can be solved in time $O(n)$ plus the time consumed by a solution for LINESEGMENTINTERSECTION, which establishes the $\Omega(n \log n)$ bound claimed.

- We could avoid identical segments by mapping $x_i$ to the line segment between the two points $(x_i, -i)$ and $(x_i, i)$.
Line Segment Intersection

**Theorem 101 (Bentley&Ottmann 1979)**

All $k$ intersections among $n$ line segments in $\mathbb{R}^2$ can be detected in $O((n + k) \log n)$ time and $O(n)$ space, using a plane-sweep algorithm.

**Corollary 102**

_LINESEGMENTINTERSECTION_ can be solved in optimal $O(n \log n)$ time and $O(n)$ space for $n$ line segments.

- Note that $n$ line segments may yield $\Theta(n^2)$ many intersections. Hence, $k \in O(n^2)$, and in the worst case the Bentley-Ottmann algorithm runs in $\Theta(n^2 \log n)$ time.

- [Chazelle&Edelsbrunner 1992] explain how to detect all $k$ intersections in $O(k + n \log n)$ time, using $O(n + k)$ space.

- [Balaban 1995] improves this to $O(k + n \log n)$ time and $O(n)$ space.
Line Segment Intersection

General position assumed

For the sake of descriptional simplicity, we assume that
- no two end-points or intersections of line segments have the same $y$-coordinate;
- no two line segments overlap;
- no three line segments intersect at the same point;
- no segment is horizontal.

- A GPA assumption makes perfect sense since it allows to avoid special cases and, thus, to focus on the essential ideas of a (geometric) algorithm.
- Different GPA assumptions might be made depending on the actual application. E.g., one might want to assume that no four points out of a given set of points in the plane are co-circular.

Caveat

A GPA assumption may make it necessary to work out all the — possibly non-trivial — subtle details on one’s own prior to an actual implementation . . .
Line Segment Intersection: Plane Sweep

- Suppose that we draw horizontal lines through all $2n$ end-points of the line segments and all $k$ intersection points.
- These $2n + k$ lines split $\mathbb{R}^2$ into $2n + k + 1$ horizontal slabs.
- Note that the left-to-right order of the line segments does not change within a slab. And this observation holds for all slabs!

Question: When does the relative order of two line segments change?

Answer: The relative order of two line segments $\ell_1, \ell_2$ changes at the border line of two adjacent slabs if that border line passes through the point of intersection of $\ell_1$ and $\ell_2$.

Lemma 103

Two line segments $\ell_1, \ell_2$ intersect if and only if there exist two adjacent slabs such that $\ell_1, \ell_2$ are neighbors in the left-to-right orders and such that the relative order of $\ell_1, \ell_2$ within the two slabs is different.
Line Segment Intersection: Plane Sweep

- **Basic idea:**
  - Sweep a horizontal line over the line segments and keep track of their left-to-right orders.
  - Halt and update these left-to-right orders whenever necessary.

### Sweep-line algorithm

A *plane-sweep algorithm* uses two data structures:

1. **Event-point schedule**: Sequence of halting positions to be assumed by the sweep line.
2. **Sweep-line status**: Description of the intersection of the sweep line with the geometric object(s) being swept at the current event.

### Sweep for line-segment intersection

Plane sweep applied to line-segment intersection detection:

1. **Event-point schedule**: End-points of all line segments of $S$ and all intersection points, arranged according to ascending $y$-coordinates. (The sweep is bottom-to-top.)
2. **Sweep-line status**: Left-to-right sequence of the line segments of $S$ that intersect the sweep line.
Line Segment Intersection: Plane Sweep

1. Initialize a priority queue $Q$ of future events:
   - Every event is associated with a point in $\mathbb{R}^2$ and with the up to two line segments on which it lies.
   - The events are prioritized according to the points’ $y$-coordinates.

2. Insert all $2n$ end-points of the $n$ line segments into $Q$.

3. Initialize a binary search tree $T$ that will contain those line segments of $S$ which are crossed by the sweep line:
   - The segments are ordered according to the $x$-coordinates of the crossing points.
   - Initially, $T$ is empty.
While $Q$ is not empty, fetch and remove the next event from $Q$. Let $p$ be the point associated with that event, and let $y_p$ be its $y$-coordinate:

- If $p$ is the lower end-point of a line segment $\ell$:
  - Insert $\ell$ into $T$.
  - Let $\ell_L$ and $\ell_R$ be the line segments that are immediately to the left and right of $\ell$, if they exist. (Use $T$ to locate $\ell_L, \ell_R$.)
  - If $\ell_L, \ell_R$ intersect above $y_p$ then remove the intersection from $Q$.
  - If $\ell_L, \ell$ or $\ell, \ell_R$ intersect above $y_p$ then insert the intersection into $Q$. 

\[
\begin{align*}
\ell_L &= c \\
\ell_R &= b \\
\ell &= d
\end{align*}
\]
4. While $Q$ is not empty, fetch and remove the next event from $Q$. Let $p$ be the point associated with that event, and let $y_p$ be its $y$-coordinate:

   a. If $p$ is the upper end-point of a line segment $\ell$:
      1. Let $\ell_L$ and $\ell_R$ be the line segments that are immediately to the left and right of $\ell$, if they exist. (Use $T$ to locate $\ell_L, \ell_R$.)
      2. Remove $\ell$ from $T$.
      3. If $\ell_L, \ell_R$ intersect above $y_p$ then insert the intersection into $Q$.

![Diagram illustrating line segment intersection using plane sweep method]
Line Segment Intersection: Plane Sweep

While $Q$ is not empty, fetch and remove the next event from $Q$. Let $p$ be the point associated with that event, and let $y_p$ be its $y$-coordinate:

1. If $p$ is a point of intersection of $\ell_1$ and $\ell_2$:
   a. Let $\ell_L$ and $\ell_R$ be the line segments that are immediately to the left of $\ell_1$ and right of $\ell_2$, if they exist. (Use $T$ to locate $\ell_L, \ell_R$.)
   b. If $\ell_L, \ell_1$ or $\ell_2, \ell_R$ intersect above $y_p$ then remove the intersection(s) from $Q$.
   c. If $\ell_1, \ell_R$ or $\ell_L, \ell_2$ intersect above $y_p$ then insert the intersection(s) into $Q$.
   d. Trade the order of $\ell_1$ and $\ell_2$ in $T$.
Line Segment Intersection: Plane Sweep

Correctness:
- Whenever two line segments \( \ell_1, \ell_2 \) are neighbors in the sorted left-to-right order of segments, the point of intersection of \( \ell_1, \ell_2 \) is present in \( Q \), if it exists and has a higher \( y \)-coordinate.
- Hence, no future event and, in particular, no point of intersection is missed.

Complexity:
- The algorithm processes a sequence of \( 2n + k \) events.
- Since future intersections between line segments are maintained in the priority queue \( Q \) if and only if the line segments currently are neighbors in the left-to-right order, at any given point in time we will never need to do maintain more than \( 3n - 1 \) events in \( Q \).
- The algorithm stores up to \( n \) line segments in left-to-right order in \( T \).
- Every event requires a constant number of updates of \( Q \) and \( T \).
- If \( Q \) and \( T \) allow insertions, deletions and searches in logarithmic time then every event is handled in \( O(\log n) \) time.
- Any standard balanced binary search tree (e.g., AVL-tree, red-black tree) and any logarithmic-time priority queue (e.g., binary heap) suffice.
- Summarizing, the Bentley-Ottmann algorithm finds all intersections among \( n \) line segments in \( O((n + k) \log n) \) time, using \( O(n) \) space.
Order Statistics, Selection and Sorting

- Order Statistics and Selection
- Linear-Time Sorting
Definition 104 (Order statistic, Dt.: Ordnungsstatistik)

Consider a finite (totally-ordered) set $S$ of $n$ distinct elements and a number $k$, for $k, n \in \mathbb{N}$. An element $x \in S$ is the $k$-th smallest element of $S$, aka the $k$-th order statistic, if $|\{s \in S : s < x\}| = k - 1$. If $k = \lceil \frac{n}{2} \rceil$ then the $k$-th smallest element of $S$ is also called the median of $S$.

Problem: 

**SELECTION**

**Input:** A set $S$ of $n$ distinct (real) numbers and a number $k$, for $k, n \in \mathbb{N}$.

**Output:** The $k$-th smallest element of $S$.

- If $k = 1$ or $k = n$ then SELECTION can be solved easily using $n - 1$ comparisons.
- If the numbers of $S$ are arranged in sorted order then the $k$-th smallest element can be found in $O(n)$ time. (Or even faster.)
- What about general values of $k$ and unsorted numbers?

Theorem 105 (Blum & Floyd & Pratt & Rivest & Tarjan 1973)

SELECTION among $n$ distinct numbers can be solved in $O(n)$ time, for any $n, k \in \mathbb{N}$.
Linear-Time Selection

Proof of Theorem 105:

1. Divide the $n$ elements of $S$ into $\lceil n/5 \rceil$ groups of 5 elements each and (at most) one group containing the remaining $n \mod 5$ elements.

Suppose that we want to compute the $k$-th smallest element of $S := \{23, 7, 15, 18, 16, 5, 64, 8, 12, 13, 11, 14, 1, 24, 6, 9, 4, 10, 3, 2, 19, 20, 21, 17\}$, for $k := 7$ and $n := |S| = 24$. 
Linear-Time Selection

*Proof of Theorem 105:*

1. Divide the $n$ elements of $S$ into $\lfloor n/5 \rfloor$ groups of 5 elements each and (at most) one group containing the remaining $n \mod 5$ elements.
2. Sort each group and compute its median.
## Linear-Time Selection

**Proof of Theorem 105:**

1. Divide the $n$ elements of $S$ into $\lfloor n/5 \rfloor$ groups of 5 elements each and (at most) one group containing the remaining $n \mod 5$ elements.
2. Sort each group and compute its median.
3. Recursively find the median $s$ of the $\lceil n/5 \rceil$ medians found in the previous step.
4. Partition $S$ relative to the median-of-medians $s$ into $S_L$ and $S_R$ (and $\{s\}$) such that all elements of $S_L$ are smaller than $s$ and all elements of $S_R$ are greater than $s$.
5. Let $m := |S_L \cup \{s\}|$. If $k = m$ then return $s$. Otherwise, if $k < m$ then recurse on $S_L$ to find the $k$-th smallest element, else (if $k > m$) recurse on $S_R$ to find the $(k - m)$-th smallest element.

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>4</th>
<th>9</th>
<th>10</th>
<th>7</th>
<th>15</th>
<th>16</th>
<th>18</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>11</td>
<td>14</td>
<td>24</td>
<td>5</td>
<td>8</td>
<td>12</td>
<td>13</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>12</td>
<td>13</td>
<td>64</td>
<td>1</td>
<td>6</td>
<td>11</td>
<td>14</td>
<td>24</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td>16</td>
<td>18</td>
<td>23</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>17</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>17</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: The elements $1, 5, 7$ and $11$ are marked as $s$ to indicate they are the medians found in the previous step.
Proof of Theorem 105 (cont’d):

What is the complexity? We get

\[ m = |S_L| + 1 \geq 3 \left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil \geq \frac{3}{10} n, \]  
and, thus, \( |S_L| \geq \frac{3}{10} n - 1 \).

Hence, \( |S_R| \leq \frac{7}{10} n \). Similarly, \( |S_R| \geq \frac{3}{10} n + O(1) \) and \( |S_L| \leq \frac{7}{10} n + O(1) \), resulting in the recurrence relation

\[ T(n) \leq T \left( \frac{n}{5} \right) + T \left( \frac{7n}{10} \right) + O(n), \]  
which yields \( T \in O(n) \). \[ \square \]
Linear-Time Selection

- Unfortunately, the constant hidden in the $O$-term is fairly large: Depending on details of the actual implementation, this algorithm requires about $50n$ comparisons!
- Hence, linear-time selection is too slow to be useful in practice.
- Open problem: Devise a linear-time algorithm for selection that has a better constant.
- What about randomization? We could pick an element of $S$ randomly and regard it as the median of medians . . .
Expected Linear-Time Selection

1. Pick an element $s$ uniformly at random from $S$.
2. Partition $S$ relative to $s$ into $S_L$ and $S_R$ (and $\{s\}$) such that all elements of $S_L$ are smaller than $s$ and all elements of $S_R$ are greater than $s$.
3. Let $m := |S_L \cup \{s\}|$. If $k = m$ then return $s$. Otherwise, if $k < m$ then recurse on $S_L$ to find the $k$-th smallest element, else (if $k > m$) recurse on $S_R$ to find the $(k - m)$-th smallest element.

What is the complexity of this randomized algorithm?

**Worst case:** If $s$ is the smallest or largest element of $S$ then $S$ shrinks by only one element, and we get $O(n^2)$ complexity.

The probability of consistently picking an element of $S$ which currently is the smallest or largest is

$$\frac{2}{n} \cdot \frac{2}{n-1} \cdot \frac{2}{n-2} \cdot \cdots \cdot \frac{2}{3} \cdot \frac{2}{2} = \frac{2^{n-1}}{n!}.$$

**Best case:** The element $s$ turns out to be the $k$-th smallest element, with probability $\frac{1}{n}$. 
Expected Linear-Time Selection

Expected complexity:

- Let $T(n)$ be an upper bound on the expected time to process a set $S$ with $n$ (or fewer) elements.
- Call $s$ lucky if $|S_L| \leq \frac{3n}{4}$ and $|S_R| \leq \frac{3n}{4}$.
- Hence, $s$ is lucky if it lies between the 25th and the 75th percentile of $S$, which happens with probability $\frac{1}{2}$.
- This gives us

$$T(n) \leq \text{(time to partition)} + \text{(maximum expected time for recursion)}$$

$$\leq n + \Pr(s \text{ is lucky}) \cdot T\left(\frac{3n}{4}\right) + \Pr(s \text{ is unlucky}) \cdot T(n)$$

$$= n + \frac{1}{2} T\left(\frac{3n}{4}\right) + \frac{1}{2} T(n).$$

- Hence, after subtracting $\frac{1}{2} T(n)$ from both sides, we get

$$T(n) \leq T\left(\frac{3n}{4}\right) + 2n,$$

i.e., $T(n) \leq 8n$.

**Theorem 106**

A simple randomized algorithm solves **Selection** in expected linear time.
Counting Sort

- Counting Sort can be used for sorting an array $A$ of $n$ elements whose keys are integers within the range $[0, k - 1]$, for some $n, k \in \mathbb{N}$.
- It is stable but not in-place.
- It uses indices into an array and, thus, is not a comparison sort.

Basic idea:

1. Compute a histogram $H$ of the number of times each element occurs within $A$.
2. For all possible keys, do a prefix sum computation on $H$ to compute the starting index in the output array of the elements which have that key.

<table>
<thead>
<tr>
<th>i, j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i, j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i, j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>H</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Counting Sort

Basic idea:
3 Move each element to its sorted position in the output array $B$.

<table>
<thead>
<tr>
<th>$i, j$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>$H$</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$B$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$i, j$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>$H$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$B$</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Theorem 107
Counting Sort is a stable sorting algorithm that sorts an array of $n$ elements whose keys are integers within the range $[0, k - 1]$, for some $n, k \in \mathbb{N}$, within $O(n + k)$ time and space.
Counting Sort

```c
CountingSort(array A[], array B[], array H[], int n, int k)
{
    /* calculate histogram */
    for (i = 0; i < k; ++i) H[i] = 0;
    for (j = 0; j < n; ++j) H[A[j]] += 1;
    /* calculate the starting index for each key */
    total = 0;
    for (i = 0; i < k; ++i) {
        oldCount = H[i];
        H[i] = total;
        total += oldCount;
    }
    /* stable copy to output array */
    for (j = 0; j < n; ++j) {
        B[H[A[j]]] = A[j];
        H[A[j]] += 1;
    }
}
```
Radix Sort

- Radix Sort can be used for sorting an array $A$ of $n$ elements whose keys are $d$-digit (non-negative) integers, for some $n, d \in \mathbb{N}$.
- It compares keys on a per-digit basis and, thus, is not a comparison sort.
- It is stable but not in-place.

Basic idea:

1. Use a stable sorting algorithm to sort the elements relative to the least significant digit of their keys.
2. Then sort on the second least-significant digit, and so on.

\[
\begin{array}{cccccccc}
21 & 123 & 234 & 34 & 23 & 923 & 863 & 950 \\
950 & 21 & 123 & 23 & 923 & 863 & 234 & 34 \\
21 & 123 & 23 & 923 & 234 & 34 & 950 & 863 \\
21 & 23 & 34 & 123 & 234 & 863 & 923 & 950
\end{array}
\]
Radix Sort: Complexity

```c
RadixSort(array A[], int n, int d)
{
    /* digit 1 is least significant,
       digit d is most significant */
    for (i = 1; i <= d; ++i) {
        use stable sort to sort A[] relative to digit i
    }
}
```

Theorem 108
Radix Sort is a stable sorting algorithm that can be implemented to sort an array of \(n\) elements whose keys are formed by the Cartesian product of \(d\) digits, with each digit out of the range \([0, k - 1]\), within \(O(d(n + k))\) time and \(O(n + k)\) space, for \(n, d, k \in \mathbb{N}\).  

**Proof:** The correctness can be established by induction. If Counting Sort is used for sorting according to the \(i\)-th digit then \(O(n + k)\) time is consumed per digit.

- It is obvious that Radix Sort can be employed whenever keys are to be sorted lexicographically such that each key is formed by the Cartesian product of “digits”, where each digit belongs to some (ordered) finite set. 

\(\square\)
Radix Sort: Discussion

- Whether or not Radix Sort is faster than comparison-based sorting algorithms depends on the assumptions made.
- If we regard an integer as a word with \( w \) bits then Theorem 108 implies that Radix Sort runs in \( O(w \cdot n) \) time, i.e., in time linear in \( n \) if \( w \) is assumed to be constant.
- However, \( w \) can hardly be considered a constant: if all \( n \) keys are distinct, then \( w \) has to be at least \( \log n \) for a RAM to be able to store them in memory, resulting in a time complexity of \( \Omega(n \log n) \).
- The \( O(n \log n) \) bound on the worst-case time complexity of optimal comparison-based sorting algorithms holds only if constant time per comparison can be assumed.
- If \( w \) cannot be assumed to be constant then this assumption is weak, too: Comparisons of randomly generated keys take constant time on average because keys differ on the very first bit in half the cases, and differ on the second bit in half of the remaining half, etc. However, in a sorting algorithm the keys cannot be regarded as random as the sort progresses!
- Radix Sort can only sort according to a lexicographical ordering, while comparison-based sorting algorithms are more general. But this tends to be of little importance in practice.
Priority Queues
- Binomial Heaps
- Fibonacci Heaps
Abstract Data Type: Priority Queue

Priority Queue (Dt.: VorrangwartenSchlange)

A priority queue (PQ) is an ADT that arranges data elements according to per-element keys ("priorities"): In a minimizing (maximizing, resp.) PQ the element with smallest (largest, resp.) overall key is served first.

- Keys need to belong to a totally ordered set.
- Standard operations for minimizing PQs:
  - FindMin: return element with smallest key,
  - DeleteMin: return and remove element with smallest key from PQ,
  - Insert: insert a new element,
  - DecreaseKey: decrease the key of an element,
  - Remove: remove an element from PQ,
  - Merge: merge (aka meld) two PQs.

- Standard implementation of PQ: binary heap.
  - FindMin in $O(1)$.
  - Insert, DeleteMin, Remove and DecreaseKey in $O(\log n)$ time if heap has $n$ elements.
  - Merge in $O(n_1 + n_2)$ time for two heaps with $n_1$ and $n_2$ elements.
Binomial Tree

Definition 109 (Binomial tree, Dt.: Binomialbaum)

A binomial tree is a rooted and ordered tree which is defined recursively as follows:

- A binomial tree of order 0 consists only of the root node;
- For $k \in \mathbb{N}_0$, a binomial tree of order $k + 1$ consists of two binomial trees of order $k$ such that one binomial tree is the left-most subtree of the other.

Lemma 110

For $k \in \mathbb{N}_0$, a binomial tree of order $k$ has $k$ subtrees (from left to right) of orders $k - 1, k - 2, \ldots, 1, 0$.

Proof: By induction on $k$.

Lemma 111

For $k \in \mathbb{N}_0$, a binomial tree of order $k$ has $2^k$ nodes and height $k$.

Lemma 112

For $k \in \mathbb{N}_0$, a binomial tree of order $k$ has $\binom{k}{d}$ nodes at depth $d$. 
Binomial Heap

**Definition 113 (Binomial heap)**

A **binomial heap** is a collection of binomial trees that satisfy the **binomial heap property**:

- Each binomial tree is a minimizing heap, i.e., for all nodes \( v \) of the binomial tree, all keys of the children of \( v \) are greater than (or at most equal to) the key of \( v \).
- For any \( k \in \mathbb{N}_0 \), there is at most one binomial tree of order \( k \).
- The binomial trees are arranged in a right-to-left sorted sequence according to their orders, with the tree of smallest order being right-most.

**Lemma 114**

For \( n \in \mathbb{N}_0 \), a binomial heap with a total of \( n \) nodes contains a binomial tree of order \( k \) if and only if the bit that corresponds to \( 2^k \) in the binary representation of \( n \) is 1.

**Proof:** Recall Lem. 111, and that the binary representation of \( n \in \mathbb{N}_0 \) is unique. 

- E.g., \( 11 = 2^3 + 2^1 + 2^0 = (1011)_2 \).
Binomial Heap: Merging in a Special Case

- Suppose that we want to merge two binomial heaps in the special case that both heaps contain only one binomial tree of the same order $k$. Let $B_1$ and $B_2$ be these two trees.
- We generate one binomial tree $B_3$ of order $k + 1$ by making $B_1$ the left-most child of $B_2$ if the key of the root of $B_2$ is less than the key of the root of $B_1$. Otherwise, $B_2$ becomes the left-most child of $B_1$.

Recap: How do we add numbers in binary representation?

E.g., let's add $n_1 := 5 = (0101)_2$ and $n_2 := 7 = (0111)_2$. We should get $12 = (1100)_2$. 

$\begin{array}{cccc}
n_1: & 0 & 1 & 0 & 1 \\
n_2: & 0 & 1 & 1 & 1 \\
\text{carry:} & 1 & 1 & 1 & - \\
\text{result:} & 1 & 1 & 0 & 0
\end{array}$
Binomial Heap: Merging

Merging Binomial Heaps

- We visit the binomial trees of both binomial heaps according to increasing order $k$, starting with $k := 0$.
- If both heaps and the carry contain exactly ...
  - ... no binomial tree of order $k$: Do nothing.
  - ... one binomial tree $B_1$ of order $k$: move $B_1$ to the result.
  - ... two binomial trees $B_1, B_2$ of order $k$: Merge $B_1$ and $B_2$ into a tree $B$ of order $k + 1$ and move $B$ to the carry.
  - ... three binomial trees $B_1, B_2, B_3$ of order $k$: Merge $B_1$ and $B_2$ into a tree $B$ of order $k + 1$ and move $B$ to the carry; move $B_3$ to the result.
- Increment $k$ after processing the binomial trees of order $k$.

Lemma 115

Merging two binomial heaps with a total of $n$ nodes takes $O(\log n)$ time.

Proof: Lemma 114 implies that a binomial heap with $i$ nodes contains at most $\lfloor \log(i) \rfloor + 1$ binomial trees. Hence, we need to perform $O(\log n)$ trivial merges of two binomial trees of the same order. Each such merge takes $O(1)$ time.
Binomial Heap: Merging

binomial heap I:

binomial heap II:

carry:

$k = 3$

merged heap:
Binomial Heap: Other Operations

**Lemma 116**

A new element can be inserted into a binomial heap with a total of $n$ nodes in $O(\log n)$ worst-case and $O(1)$ amortized time.

**Proof:** We create a new heap that contains only the new element and merge it with the old heap. The amortized analysis is similar to the one used for incrementing a binary counter.

**Lemma 117**

Finding the minimum element in a binomial heap with a total of $n$ nodes takes $O(\log n)$ time.

**Proof:** It suffices to inspect the roots of all binomial trees of the heap.

- By maintaining a pointer to the root with minimum key, this time can be reduced to $O(1)$. (The pointer can be updated during all operations without increasing the complexity bounds.)
Binomial Heap: Other Operations

**Lemma 118**

The minimum element can be deleted from a binomial heap with a total of $n$ nodes in $O(\log n)$ time.

*Proof:* We find the minimum among the roots of the binomial trees. By removing this root we split one binomial tree into a sequence of subtrees which in turn are binomial trees and, thus, form a binomial heap. Now we merge this new binomial heap with the rest of the original binomial heap. All these steps run in $O(\log n)$ time.

**Lemma 119**

An element can be deleted from from a binomial heap with a total of $n$ nodes in $O(\log n)$ time.

*Proof:* We first decrease the key of the element to a value smaller than the minimum key contained in the heap, thus causing it to move upwards to a root, and then delete that root.
Lemma 120

The key of a known element of a binomial heap with a total of $n$ nodes can be decreased in $O(\log n)$ time.

Proof: After decreasing the key we may need to (repeatedly) exchange the corresponding node with its parent node if the min-heap property is violated. Since any binomial tree of the heap has height at most $\log n$, the claim follows.

- Note: With the exception of the $O(1)$ bound on the amortized time needed for one insert, all other time bounds are worst-case bounds!
Fibonacci Heaps: Basics

- Designed by Fredman and Tarjan in 1986, in an attempt to improve Dijkstra’s shortest path algorithm from $O(|E| \log |V|)$ to $O(|E| + |V| \log |V|)$.
- The name is derived from the fact that the Fibonacci numbers show up in the complexity analysis of its operations.
- Similar to binomial heaps, but less rigid: Fibonacci heaps *lazily* defer all clean-up work after an Insert till the next DeleteMin.

Fibonacci Heap

- Collection of min heaps.
- Maintains pointer to element with minimum key.
- Some nodes are “marked”. (Used to keep trees reasonably flat.)
Fibonacci Heaps: Representation

Heap representation:
- Maintain root nodes in doubly-linked circular list.
- Store pointer to root node with minimum key.

Node representation: Every node stores:
- A pointer to its parent.
- A pointer to one of its children.
- The number of its children (“order”, “rank”).
- Pointers to its left and right siblings.
- A binary flag that indicates whether the node is marked (indicated by gray shading).
Fibonacci Heaps: Marked Nodes

- Marking of nodes:
  - **Unmarked**: The node has had no child cut.
  - **Marked**: The node has had one child cut.

- Basic idea: When a child is cut from a marked parent node, then the parent node (together with its entire subtree) is cut, too, and moved to the root list.

- The marking of nodes ensures that Fibonacci heaps keep roughly the structure of binomial heaps after the deletion of nodes, thus ensuring the amortized time bounds.

- A root node is always unmarked.
**Fibonacci Heaps: Basic Operations**

**Insert** a new node:
- Create a new node and insert it into the list of root nodes.
- Update pointer to (new) minimum root node.

**Link** two trees with roots $r_1$ and $r_2$:
- If $r_1.key \geq r_2.key$ then make $r_1$ a child of $r_2$; otherwise, $r_2$ becomes a child of $r_1$.
- Update information on the order of $r_2$ (or $r_1$).

**Cut** a node $v$ (that is not a root node):
- Remove $v$ (and its subtree) from the child list of its parent $p$ and insert it into the root list.
- Update information on the order of $p$.
- Mark $p$. 

![Diagram](image-url)
Fibonacci Heaps: DeleteMin

DeleteMin:
- Delete the root node with the current minimum.
- Move its children as new root nodes into the list of root nodes.
- Link trees until no pair of root nodes has the same order.
- Update pointer to minimum root.

First DeleteMin. Second DeleteMin.
Fibonacci Heaps: DecreaseKey

DecreaseKey:
- If the new key of $v$ is less than the key of the parent $p$ then cut $v$ and move it (with its subtree) to the root list.
- If $p$ is not marked then mark $p$.
- Else, cut $p$ and move to root list, and apply recursively to its parent.
- Update pointer to minimum root.

DecreaseKey(9,6).
Lemma 121

If only Insert and DeleteMin operations are carried out, then a Fibonacci heap is a binomial heap after every DeleteMin operation.

**Sketch of Proof:** By induction: Every DeleteMin results in a consolidation phase during which pairs of trees which have root nodes of the same order are linked.

- If no consolidation occurs (since no suitable DeleteMin operation is carried out) then a Fibonacci heap with \( n \) nodes may degenerate to one single tree, or even to an unsorted linked list (of \( n \) root nodes) or an “unary” tree of height \( n - 1 \).
Fibonacci Heaps: Properties

**Lemma 122**

If a node of a tree in a Fibonacci heap has \( k \) children then it is the root of a subtree with at least \( F_{k+2} \) nodes.

**Corollary 123**

Every node of a tree in a Fibonacci heap with a total of \( n \) nodes has at most \( O(\log n) \) children.

**Proof:** Let \( k \) be the number of children of a node \( v \). By Lem. 122, its subtree has \( F_{k+2} \) nodes. Hence,

\[
 n \geq F_{k+2} \geq \phi^k, \quad \text{implying} \quad k \leq \log_\phi n. \]

\( \square \)
Fibonacci Heaps vs. Binomial Heaps

Theorem 124

When starting from an initially empty heap, any sequence of $a$ Insert, $b$ DeleteMin and $c$ DecreaseKey operations takes $O(a + b \log n + c)$ worst-case time, where $n$ is the maximum heap size.

- Hence, from a theoretical point of view, a Fibonacci heap is better than a binomial heap when $c$ is smaller than $b$ by a non-constant factor.
- A Fibonacci heap is also better than a binomial heap when frequent merging of heaps is required.
- However, the worst-case time for one DeleteMin or DecreaseKey operation is linear, which makes Fibonacci heaps less suitable for applications which cannot tolerate excessive running times for one individual operation. (E.g., real-time systems.)
- There is some controversy about Fibonacci heaps: While some researcher strongly advocate their use, other report Fibonacci heaps to be slow in practice, due to hidden constants in the $O$-terms.
### Performance Summary for Priority Queues with $n$ Elements

<table>
<thead>
<tr>
<th>Operation</th>
<th>Linked List</th>
<th>Binary Heap</th>
<th>Binomial Heap</th>
<th>Fibonacci Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(1)^*$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>FindMin</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(\log n)^{**}$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>DeleteMin</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)^{**}$</td>
</tr>
<tr>
<td>DecreaseKey</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(1)^{**}$</td>
</tr>
<tr>
<td>Merge</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

*: amortized complexity; worst-case complexity is $O(\log n)$.

**: amortized complexity; worst-case complexity is $O(n)$.

***: can be brought down to $O(1)$ with little extra effort.

- Note: Attempts to get Insert, DeleteMin and DecreaseKey all down to $O(1)$ are doomed to fail. (At least as long as we allow only key comparisons.)
Randomized Data Structures for Searching
- Basics: Dictionary, Set and Searching
- Randomizing Binary Search Trees
- Treaps
- Skip Lists
- Hashing
Abstract Data Type: Dictionary

**Dictionary (Dt.: Wörterbuch)**

A *dictionary* is a collection ADT that focuses on data storage and retrieval; i.e., it is a searchable structure.

- Data is a *key-value pair* (KVP), a so-called *item*: \((k, v)\).
- Standard operations:
  - Insert item \((k, v)\) into structure,
  - Retrieve data from structure, i.e., check whether it has an item with a given key \(k\) and return the pair \((k, v)\),
  - Delete item from structure.

- In addition, a reassign replaces the value in one of the \((k, v)\) pairs in the structure. Also: Predecessor, successor, join . . .

- Multiple entries with the same key may or may not be allowed. Unless stated otherwise, we assume all keys to be distinct.
- Related terms and synonyms: Key-value database, associative array, map.
- Since the values of the key-value pairs of a dictionary are there only for a piggyback ride, we simply omit the values in the figures and pseudo codes.
Abstract Data Type: Set

Set (Dt.: Menge)

A set is a collection ADT that allows to store data items and focuses on efficient membership tests.

- **Data items:**
  - A data item is a key or key-value pair with trivial value.
  - The order of the data items does not matter (or is undefined),
  - Duplicate data items are not permitted.

- **Standard operations:**
  - Insert item into structure,
  - Delete item from structure,
  - Test membership, i.e., check whether it has an item with a given key $k$.

- **Core set-theoretic operations for two sets $S$, $T$:**
  - Union: Compute the union of $S$ and $T$,
  - Intersection: Compute the intersection of $S$ and $T$,
  - Difference: Compute the difference of $S$ and $T$,
  - Subset: Check whether $S$ is a subset of $T$.

- If duplicate items are allowed: *multiset* or *bag*. 
Complexity of Searching

Theorem 125

Comparison-based searching among $n$ elements requires $\Omega(\log n)$ comparisons in the worst case.

Proof: Assume that we want to search for the item that has key $k$ among the items $a_1, a_2, \ldots, a_n$. A decision tree $T$ for solving this problem must contain at least $n + 1$ leaves:

- one leaf for each $a_i$, if $k$ is the key of $a_i$,
- one additional leaf for “not found”.

Hence, the height of $T$ is at least $\log(n + 1) \in \Omega(\log n)$.

- We want $O(\log n)$ running time for dictionary operations, where $n$ is the number of items in the dictionary.
- Worst case? Average case? Amortized? Special keys?
- Huge number of results known! We can barely scratch the surface . . .
Balanced Binary Search Trees

Definition 126 (Height, Dt.: Höhe)

The *height* of a rooted tree $T$ is the maximum depth of nodes of $T$, with (by convention) the root of $T$ being at depth 0.

Definition 127 (Balanced tree, Dt.: balanzierter Baum)

A binary tree is *height-balanced* if it either has no proper subtrees or if

1. it has two proper subtrees and the heights of both subtrees differ by not more than 1, or if
2. it has exactly one proper subtree of height 0,

and if
3. all proper subtrees are height-balanced.

Theorem 128

If $T$ is a balanced binary tree with $n$ nodes and height $h$ then $h \in \Theta(\log n)$.

- If $T$ remains balanced after insertions/deletions then it is called *self-balancing*. 
Balanced Binary Search Trees: AVL Trees and Friends

- [Adelson-Vel’skii&Landis 1962]: First self-balancing binary search tree (BST).
- Rotations to re-balance the tree after insertion or deletion.
- Insertion, searching and deletion all take $O(\log n)$ time in both the average case and the worst case, where $n$ is the number of nodes in the tree.
- AVL insertions require $O(1)$ rotations, while deletions require $O(\log n)$ rotations in the worst case. (But also $O(1)$ on average.)
- Height is at most $\frac{1}{\log \phi} \log n \approx 1.440 \log n \approx 2.077 \ln n$, with $\phi := \frac{1+\sqrt{5}}{2} \approx 1.618$.

- Red-black trees: Have a larger height of at most $2 \log n$, but tend to use fewer rotations. Since AVL trees are more rigidly balanced than red-black trees, they tend to have slower insertion and deletion but faster search.
- Scapegoat trees: Have no additional per-node memory overhead; provide worst-case $O(\log n)$ search time, and $O(\log n)$ amortized insertion and deletion time.
Balanced Binary Search Trees: AVL Trees and Friends

- Insertion of 8 into sample AVL tree. Rotation to re-balance.

Can we relax the balancing schemes?

Do we need the overhead caused by balancing BSTs? What could be modified?
A randomly built binary search tree with $n$ nodes is a binary search tree built by inserting $n$ items/keys in random order.

- Of course, this is equivalent to computing a random permutation of the items — where every permutation is equally likely! — and then inserting the items in that order.
- Different permutations may result in the same tree.
- This is different from assuming that every binary search tree is equally likely to occur.
- It depends on the application whether randomness can be assumed. Otherwise, the resulting tree could be highly skewed.
- What is this good for?
  - Well, if you insert 10 numbers in random order then the resulting tree will degenerate to a list with probability $\frac{2}{10!} \approx 5.511 \cdot 10^{-7}$.
  - The more nodes, the less likely the tree is degenerate: It is non-degenerate with high probability.
**Randomly Built Binary Search Trees**

**Lemma 129**
The expected time to randomly build a binary search tree with $n$ nodes is $O(n \log n)$.

*Sketch of Proof:* During the construction of a randomly built BST we perform the same comparisons as a randomized QuickSort, but in a different order. Hence, Theorem 91 is applicable and we also get an $O(n \log n)$ expected-time bound.

![Graphical representation of a binary search tree]

- Hence, one can also sort in expected $O(n \log n)$ time by constructing a randomly built binary search tree and then applying an inorder traversal.
Randomly Built Binary Search Trees: Node Depth

Lemma 130
The average node depth of a randomly built binary search tree is $O(\log n)$.

Proof: The depth of a node equals the number of comparisons made during the BST construction. Since all permutations of the keys are equally likely, the average node depth $d_n$ is given by

$$d_n = \frac{1}{n} E \left[ \sum_{i=1}^{n} (\# \text{ comparisons for node } i) \right] = \frac{1}{n} O(n \log n) = O(\log n).$$

Even if the average depth of a node is $\Theta(\log n)$, the height of its tree can still be $\omega(\log n)$.

Theorem 131 (Reed 2003)
A randomly built binary search tree with $n$ nodes has an expected height of $\alpha \ln n$, where $\alpha := 4.311\ldots$ is the unique solution within $[2, \infty)$ of the equation $\alpha \ln\left(\frac{2e}{\alpha}\right) = 1$.

Little is known if both insertions and deletions are allowed. Deletions destroy the randomness, and experiments suggest that the height will tend to $O(\sqrt{n})$. 
Randomized Binary Search Trees

Randomized Binary Search Tree

A binary search tree $T$ with $n$ nodes is a randomized binary search tree (RBST) if either $n = 0$ or if, for $n > 0$,

1. both its left subtree $L$ and right subtree $R$ are independent randomized binary search trees,

2. $\Pr(L \text{ has } i \text{ nodes}) = \frac{1}{n}$ for all $0 \leq i \leq n - 1$.

- Invented by Martínez and Roura (1998).
- The randomization implies that every item has the same probability of $\frac{1}{n}$ to be at the root of the tree.
- In an implementation: Pick a random integer $k$ with $0 \leq k \leq n$, where $n$ is the current number of nodes of $T$. If $k = n$ then insert at the root of $T$; otherwise insert recursively into the proper subtree of $T$.

Theorem 132

The expected height of a randomized binary search tree with $n$ nodes is $O(\log n)$. 
Randomized Binary Search Trees: Rotations

- Simple left and right rotations are carried out in order to maintain the property of being a BST.
- A rotation decreases the depth of one node and increases the depth of another node by one.
- Rotations can be performed in $O(1)$ time because they involve only simple pointer manipulations.
Randomized Binary Search Trees: Rotations

```
rotateRight(node u, bst T)
{
    node w = u.lft;
    w.parent = u.parent;
    if (u != T.root) {
        if (u.parent.lft == u) u.parent.lft = w;
        else u.parent.rgt = w;
    }
    u.lft = w.rgt;
    if (u.lft != NIL) u.lft.parent = u;
    u.parent = w;
    w.rgt = u;
    if (u == T.root) T.root = w;

    return;
}
```
Randomized Binary Search Trees: Rotations

rotateLeft(node w, bst T)
{
    node u = w.rgt;
    u.parent = w.parent;
    if (w != T.root) {
        if (w.parent.lft == w) w.parent.lft = u;
        else w.parent.rgt = u;
    }
    w.rgt = u.lft;
    if (w.rgt != nil) w.rgt.parent = w;
    w.parent = u;
    u.lft = w;
    if (w == T.root) T.root = u;

    return;
}
Randomized Binary Search Trees: Insertion

```c
randomizedInsert(key x, bst T)
{
    pick a random number, k, between 0 and T.size, inclusive;
    if (k == T.size) {
        insertAtRoot(x, T);
    }
    else {
        if (x < T.key) T.lft = randomizedInsert(x, T.lft);
        else T.rgt = randomizedInsert(x, T.rgt);
    }
}

insertAtRoot(key x, bst T)
{
    use standard BST algorithm to insert x as a leaf in T;
    perform left/right rotations to move the node containing x all the way up to the root of T;
}
```
Randomized Binary Search Trees: Sample Insertion at Root

1. Insert new key 6 into sample RBST: As in a standard BST, let key 6 trickle down to appropriate leaf.

2. Move new leaf node that stores 6 upwards, thereby performing left and right rotations:
   right rotation, left rotation, left rotation, right rotation, right rotation.

![Binary Search Tree Diagrams](image-url)
Randomized Binary Search Trees: Deletion

- We make use of a *join* operation for two RBSTs $L$ and $R$, where all keys in $L$ are assumed to be less than all keys in $R$:
  - Let $n_L$ be the size of $L$, and $n_R$ be the size of $R$.
  - Use root of $L$ as root of the union tree with probability $\frac{n_L}{n_L + n_R}$, and recursively join right subtree of $L$ with $R$.
  - Use root of $R$ as root of the union tree with probability $\frac{n_R}{n_L + n_R}$, and recursively join $L$ with left subtree of $R$.

  - Deletion:
    - Search and delete the node that contains the key sought.
    - Use join operation to join the two subtrees of that node.

Lemma 133

Tree is still random after deletion.

Theorem 134

The expected height of a randomized binary search tree with $n$ nodes is $O(\log n)$. Search, insertion, deletion and join all run in $O(\log n)$ expected time.
Randomized Binary Search Trees: Deletion

```c
randomizedJoin(bst L, bst R)
{
    pick a random number, k, between 1 and (L.size + R.size);
    if (k <= L.size) {
        T = L;
        T.rgt = randomizedJoin(L.rgt, R);
    }
    else {
        T = R;
        T.lft = randomizedJoin(L, R.lft);
    }
}

delete(key x, bst T)
{
    search node N such that N.key equals x;
    randomizedJoin(N.lft, N.rgt);
    remove(N);
}
```
Treaps

Treap

A treap is a binary tree in which every node stores a priority in addition to the key-value pair such that

- it is a binary search tree on the keys,
- it is a max-heap on the priorities, where greater number means higher priority.

- All keys and priorities are assumed to be distinct.
- In the figures we use letters for the search keys and integers for the priorities.

Lemma 135

The structure of a treap is completely determined by the search keys and priorities of its nodes.

**Sketch of Proof**: We use induction. The base case is the treap with at most one node. Since a treap is a heap, the node with highest priority must be at its root. Since a treap is a BST, all nodes in its left subtree need to have keys less than the key of the root, and all nodes in its right subtree need to have keys greater than the key of the root. Since subtrees are treaps themselves, their structure is completely determined by the inductive hypothesis.
Treaps: Construction

- The proof of Lemma 135 suggests a way to construct a treap for a given set of key-(value-)priority triples:

![Treap Diagram]

- Treaps were invented by Vuillemin (1980), and rediscovered and used as RBSTs by Aragon and Seidel in 1989.
Treaps: Alternate Characterization

- Alternate characterization of treaps, with proof by induction:
  1. Sort nodes by priority.
  2. Insert one node at a time into BST according to key.

- Yet another geometric characterization:
  1. Regard key-priority pairs as coordinates in $\mathbb{R}^2$.
  2. Recursively split (portions of) the plane by inserting T-shaped boundaries.
Treaps: Operations

Search:
- Since a treap is a BST, we can apply the algorithm for searching in a BST.

Insertion:
- We use the algorithm for insertion into a BST, thus creating a node $z$.
- In order to repair the heap structure, we use rotations to “bubble” $z$ upwards as long as $z$ has a greater priority than its parent.

Deletion:
- Search the node $z$ sought.
- Use rotations to push $z$ downwards until it becomes a leaf, thereby moving its higher-priority child upwards. (Inverse rotations as for insertion!)

Split: Split treap $T$ into two treaps $T_1$, $T_2$ such that all keys of $T_1$ are less than some given key $x$ and all keys of $T_2$ are greater than $x$.
- Insert a dummy node with key $x$ and priority $+\infty$ into $T$.
- This node will become the root of the new treap, and its subtrees form the two treaps $T_1$, $T_2$ sought.

Lemma 136
The cost of each of these operations is proportional to the height of the treap.
Treaps: Sample Insertion

- Insertion of item with key $S$ and priority $9$: Create new leaf node at appropriate place. Left rotation to bubble up. Right rotation to bubble up.
Treaps: Sample Deletion

- Deletion of node with key $S$: Left rotation to push node down. Right rotation to push down. Deletion of node.
Treaps: Operations

```c
bubbleUp(node z, treap T) {
    while ((z.parent != NIL) && (z.parent.p > z.p)) {
        node u = z.parent;
        if (z.parent.rgt == z) rotateLeft(z.parent, T);
        else rotateRight(z.parent, T);
        z = u;
    }
    if (z.parent == NIL) T.root = z;
    return;
}
```
Treaps: Randomization

### Randomized Treap

A *randomized treap* is a treap in which the priorities are independently and uniformly distributed continuous random variables.

- Typically, the term “treap” has come to mean almost exclusively “randomized treap”, and it is common to drop the word “randomized”.
- When inserting a new key-value pair we generate a random real number between, e.g., 0 and 1, and use that number as the priority of the new node.
- By using reals as priorities we ensure that the probability of two nodes having equal priority is zero. In practice, choosing a random integer from a large range or a random floating-point number is good enough.
- Since the priorities are independent, each node is equally likely to have the largest priority and, thus, to be at the root of the treap.
- Hence, a *(randomized) treap is a randomized binary search tree!* Lemma 136 implies the following main result. (A formal proof is very similar to RBSTs.)

### Theorem 137

The expected height of a treap with \( n \) nodes is \( O(\log n) \). Search, insertion, deletion and split all run in \( O(\log n) \) expected time. The expected number of rotations done during an insertion/deletion is only \( O(1) \).
## Sorted Linked List

### Pros
- Truly simple dynamic data structure that is easy to implement.
- No need for an a-priory estimate of the number of elements to be stored.
- Easy to insert or delete in $O(1)$ time if position is known.

### Cons
- Difficult to get to the middle of the list; binary search does not work.
- Search in $o(n)$ time is difficult.

- Goal: Combine the appealing simplicity of sorted lists with a good expected-time behavior!
Perfect Skip Lists

- Idea: Add a second list $L_1$ containing only every second item. Then we need at most $\lceil \frac{1}{2} n \rceil$ comparisons on $L_1$ and, with proper links into the first list $L_0$, one additional comparison on $L_0$.

- If $k$ nested lists are used: At most $\lceil \frac{1}{2^{k-1}} n \rceil$ comparisons on the $k$-th list $L_{k-1}$, plus one additional comparison in each of the lists $L_0, L_1, \ldots, L_{k-2}$.

- This will get us $O(\log n)$ search time for $k := O(\log n)$.

- Header and sentinel nodes are in every level.

- Nodes are of variable size: Contain between 1 and $O(\log n)$ pointers.

- The number of pointers does not change after an insertion; can use array to store the pointers.
Perfect Skip Lists: Search

- To search for an item given a query key, we start on the list at the top level.
- In the current list we move towards the sentinel until the key of the next item will be greater than the query key.
- Then we go down and repeat the procedure, until we are in the bottom list $L_0$.
- In the bottom list $L_0$ we either find the item queried, or no such item exists.
- When searching for $k$:
  - If $k = \text{next}.k$: done!
  - If $k > \text{next}.k$: go right. Stop at sentinel.
  - If $k < \text{next}.k$: go down one level from $L_i$ to $L_{i-1}$. Stop at $L_0$.
- $O(\log n)$ levels, and will visit at most 2 nodes per level: $O(\log n)$ search time.
Perfect Skip Lists: Search

- The following sample code for a search in a skip list assumes the existence of a sentinel key that is guaranteed to be greater than any search key.

```c
searchSkipList(key x, skiplist T)
{
    Node u = T.header;
    int h = T.height;
    while (h >= 0) {
        while (u.next[h].key < x) /* assumes sentinel */
            u = u.next[h];
        --h;
    }
    return u;
}
```

---

© M. Held (Univ. Salzburg)  
Advanced Algorithms and Data Structures (WS 2019/20)
Skip Lists

- Maintaining perfect skip lists after insertions and deletions may require re-arranging the entire structure ...
- Goal: Design a hierarchical structure of singly-linked lists such that we can expect about $\frac{1}{2}$ the items at the next higher level.
- [Pugh 1989:] Probabilistic skip lists refine the idea of using a linked hierarchy of sublists by dropping the constraint that lists jump a number of items that equals a power of two.
- Skip lists are a randomized data structure which achieve expected $O(\log n)$ complexity for search, insert and delete operations.

  “Skip lists are a probabilistic data structure that seem likely to supplant balanced trees as the implementation method of choice for many applications. Skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster and use less space.”  

- Skip lists are “better trees”, but remain about as easy to implement as standard sorted linked lists.
- Actual timings for the same sequence of operations may vary depending on the random choices made by the data structure.
Skip Lists: Insertion and Removal

- Allow for some imbalance. Still, the expected behavior (over the random choices) shall remain the same as with perfect skip lists.

- Insertion:
  - Insert item into full list $L_0$, i.e., at the lowest level 0.
  - Promote it to the next higher level with (independent) probability $p$.

- Common choices for $p$ are $\frac{1}{2}$ and $\frac{1}{4}$.

- Choice of $p$ allows a trade-off between space complexity and query speed.

- We focus on $p = \frac{1}{2}$: Then the highest level of a newly inserted item can be determined by repeatedly flipping a coin until the coin comes up heads.

- Level structure of a skip list is independent of the keys inserted. The expectation is over the random coin flips. Hence, there are no “bad” key sequences that might cause a skip list to degenerate.

- Deletion: Search and then remove item from structure.

- With some very small probability, the skip list will just be a linked list, or the skip list will have every node at every level.

- Note: Parallel insertions or deletions are relatively easy to support!
Skip Lists: Analysis

- The height of a skip list is the number of its levels, with the bottom-most list $L_0$ at level 0.

Lemma 138

The expected number of times a fair coin is tossed up to and including the first time the coin comes up heads is 2.

Proof: Let $T$ denote this random variable and define an indicator random variable $I_i$ as $I_i := 1$ if the coin ends up being tossed $i$ or more times, and $I_i := 0$ otherwise. We have

$$
\mathbb{E}(I_i) = \Pr(I_i = 1) = \frac{1}{2^{i-1}}
$$

and

$$
T = \sum_{i=1}^{\infty} I_i.
$$

This gives

$$
\mathbb{E}(T) = \mathbb{E}\left(\sum_{i=1}^{\infty} I_i\right) = \sum_{i=1}^{\infty} \mathbb{E}(I_i) = \sum_{i=1}^{\infty} \frac{1}{2^{i-1}} = \sum_{i=0}^{\infty} \frac{1}{2^i} = 2.
$$

Hence, the expected number of levels for a newly inserted item is 2!
Skip Lists: Analysis

Lemma 139

Suppose that we use constant-size nodes, with one node per level if an item is stored in that level. Then the expected number of nodes in a skip list storing $n$ items is $2n$ if we disregard header and sentinel nodes.

**Proof:** The probability of an item to be included in list $L_i$ is $1/2^i$. Therefore the expected number of nodes in $L_i$ is $n/2^i$ and we get

$$
\sum_{i=0}^{\infty} \frac{n}{2^i} = n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n
$$

as the total expected number of nodes.

- Hence, linear storage can be expected to suffice for storing a skip list.
Skip Lists: Analysis

Lemma 140
The expected height of a skip list storing \( n \) items is at most \( \log n + 2 \).

Lemma 141
The expected length of a search path in a skip list storing \( n \) items is \( 2 \log n + 2 \).

Theorem 142
A skip list storing \( n \) items has expected size \( O(n) \) and supports search, insertion and deletion in expected time \( O(\log n) \).
**Insertion:** To insert item $k$ with key $x$ we
- pick a height $h$ for $k$ by flipping coins,
- create a node for $k$ with space for $h$ next pointers,
- follow the search path for $x$ downwards: if $i \leq h$ then we insert $k$ into $L_i$ by straightforward splitting and splicing.

**Deletion:** To delete item $k$ with key $x$ we
- follow the search path for $x$ downwards: when the node containing $k$ is immediately to the right then we splice out that node.
Skip Lists: Implementational Issues

```c
insertProbabilisticSkipList(key x, skiplist T) {
Node u = T.header;
int h = T.height;
int hx = result of coin flips;
Node v = CreateNode(x, hx);
while (h >= 0) {
    while (u.next[h].key < x) /* assumes sentinel */
        u = u.next[h];
    if (h <= hx) {
        v.next[h] = u.next[h];
        u.next[h] = v;
    }
    --h;
}
++T.counter_of_nodes;
return v;
}
```
Skip Lists: Implementational Issues

def deleteProbabilisticSkipList(key x, skiplist T):
    Node u = T.header;
    int h = T.height;
    boolean removed = false;
    Node v = CreateNode(x, hx);
    while (h >= 0) {
        while (u.next[h].key < x) /* assumes sentinel */
            u = u.next[h];
        if (u.next[h].key == x) {
            removed = true;
            u.prev[h].next[h] = u.next[h]; /* can avoid prev */
        }
        --h;
    }
    if (removed) --T.counter_of_nodes;
    return removed;
Skip Lists: Extension

- We can count the number of edges in a search path, in order to gain access to the $j$-th item stored in the skip list:
  - length of edge in $L_0$ is 1,
  - length of edge in $L_i$ is the sum of the lengths of the edges in $L_{i-1}$ below it.
- To get to the $j$-th node we
  - go right if the sum of the edge lengths so far plus the length of the next edge is less than $j$,
  - go down otherwise.
- This makes it easy to get the $j$-th item in the sorted list in $O(\log n)$ time, and to set/modify its value. Within the limits imposed by the fact the sequence has to remain sorted, we can even modify its key.
- Faster get/set than linked list; faster add/delete than array-based list.
Skip Lists: Rope as a Sample Application

Rope

A rope (aka cord) is a data structure that is used to efficiently store and manipulate a very long string of characters by maintaining a number of short (sub-)strings. Operations like insertion, deletion, and random access shall be supported efficiently.

- Used in many applications that maintain long strings which change over time. E.g., word processors, text editors.
- Let $R := s_0s_1 \ldots s_{n-1}$ be a string of length $n$ that is stored as a rope, and let $R_1, R_2$ be rope strings.
- Rope operations:
  - $\text{concat}(R_1, R_2)$: Return a rope that contains the concatenation of the strings $R_1, R_2$.
  - $\text{split}(R, i)$: Truncate $R$ to length $i$, i.e., $s_0s_1 \ldots s_{i-1}$, and return a rope that contains the remaining characters, i.e., the string $s_is_{i+1} \ldots s_{n-1}$.
  - $\text{report}(R, i, j)$: Return a rope that contains the string $s_is_{i+1} \ldots s_{j-1}$.
  - $\text{insert}(R, R_1, i)$: Return rope with string $R_1$ inserted at position $i$ of $R$.
  - $\text{delete}(R, i, j)$: Return rope without the string $s_is_{i+1} \ldots s_{j-1}$ of $R$.
- Insertion and deletion can be realized by means of the first three operations.
Skip Lists: Rope as a Sample Application

- We can split a skip list at any node in $O(\log n)$ expected time:
  1. Insert a new sentinel and new header at the desired position.
  2. Cut old pointers and set up new pointers.
- We can also join two skip lists with $n_1$ and $n_2$ nodes in $O(\log(\max\{n_1, n_2\}))$ expected time:
  1. Set up new pointers.
  2. Delete sentinel and header.
- Rope as skip list:
  - The nodes are implemented such that constant-time splits are possible for (short) strings, e.g., as array-based “splittable strings”.
  - As edge length we use the total length of all strings under the edge.
  - Ropes implemented as skip lists support
    - concatenation, split, insert and delete in logarithmic expected time,
    - report in logarithmic expected time plus output-sensitive $O(j - i)$ time to copy the substring of length $j - i$. 
Can we realize a data structure that supports insert, retrieve and delete operations in $O(1)$ time?

Suppose that every key-value pair has a key drawn from the universe $U := \{0, 1, \ldots, n - 1\}$, for some $n \in \mathbb{N}$. (i.e., $U = \mathbb{Z}_n$.)

In order to represent a dynamic set $S$ of KVPs we could use a *direct-address table* of size $n$, denoted by $T[0, 1, \ldots, n - 1]$:

If $S$ contains the key-value pair $(k, v)$ then we store a pointer to $v$ in $T[k]$, and $NIL$ otherwise.

![Diagram of direct addressing](image-url)
Direct Addressing

- Direct Addressing: The standard dictionary operations insert, retrieve and delete are trivial to implement, provided that all key-value pairs have distinct keys.
- Each operation runs in $O(1)$ time.
- Obvious drawbacks:
  - If $|U|$ is large then it may be impractical or even impossible to store a table of size $|U|$.
  - If $|S| \ll |U|$, then allocating a table of size $|U|$ is a waste of memory.

- Can we trade $O(1)$ worst-case complexity for $O(1)$ average-case complexity and reduce the memory requirement to $\Theta(|S|)$?
Basics of Hashing

A hash function, \( h : U \rightarrow \mathbb{Z}_m \), maps a key \( k \) of the universe \( U \) to the slot (aka bucket) \( h(k) \) of the hash table \( T[0, 1, \ldots, m - 1] \), for \( m \in \mathbb{N} \).

- We say that \( k \) hashes to \( h(k) \), and \( h(k) \) is the hash value of \( k \).
- Pick appropriate \( m \) and use hash function that can be evaluated in constant time.
Basics of Hashing

Hashing

A hash function, $h: U \rightarrow \mathbb{Z}_m$, maps a key $k$ of the universe $U$ to the slot (aka bucket) $h(k)$ of the hash table $T[0, 1, \ldots, m - 1]$, for $m \in \mathbb{N}$.

- If $m < |S|$ then the pigeonhole principle implies that at least two keys will hash to the same slot, for any hash function $h$. Could happen also for $m \geq |S|$!
- Such a situation is called a collision, which we need to resolve.

Standard methods for resolving collisions:
- Chaining: Use a list for $T[h(k)]$.
- Open addressing: Allow alternate slots instead of $h(k)$. Lazy deletion; insertion is $\Theta(1)$ only on average.
Resolving Collisions: Separate Chaining

- Chaining: Rather than letting $h(k)$ point to a single memory cell that stores $v$, we let it point to a list which contains all KVPs whose keys hash to the same slot.
- Dt.: Hashing mit Verkettung.

![Diagram of a hash table with separate chaining](image)

- $U$ and $S$ represent the hash function and its output, respectively.
- The hash table $T$ contains the KVPs $8 \text{ v}_8$, $2 \text{ v}_2$, $4 \text{ v}_4$, and $5 \text{ v}_5$.
- The elements are distributed across slots 0, 1, 2, and 3, with collisions resolved by separate chaining.
Resolving Collisions: Separate Chaining

- Chaining: Rather than letting $h(k)$ point to a single memory cell that stores $v$, we let it point to a list which contains all KVPs whose keys hash to the same slot.
- Then insertion maintains its $O(1)$ worst-case complexity (if we may assume that the KVP to be inserted is not yet present in the hash table).
- The complexity of retrieving a KVP depends on the (maximum) length of a list.
- Worst-case complexity of retrieval: $\Theta(n)$ if $n$ KVPs have been stored in the table in a single list.
- An actual deletion of a KVP (upon its prior location) can be done in $O(1)$ time (if doubly-linked lists are used).
- Denote the length of the list referenced by $T[i]$ by $n_i$. We have
  $$n_0 + n_1 + \cdots + n_{m-1} = n.$$ 
- Can we say anything on $E(n_i)$ and, thus, on the expected complexity of a search?
Resolving Collisions: Separate Chaining

**Definition 143 (Load factor)**

Let a hash table $T$ store $n$ KVPs in a total of $m$ slots. The *load factor* $\alpha$ of $T$ is defined as

$$\alpha := \frac{n}{m}.$$ 

**Uniform hashing**

A key $k$ is equally likely to hash into any of the $m$ slots, independently of where any other key has hashed to.

Hence, with uniform hashing we have

$$\Pr(h(k) = i) = \frac{1}{m} \quad \text{for all } i \in \{0, 1, \ldots, m-1\} \text{ and all } k \in U,$$

and

$$\mathbb{E}(n_i) = \sum_{j=0}^{n-1} 1 \cdot \Pr(h(k_j) = i) = \frac{n}{m} = \alpha.$$
Resolving Collisions: Separate Chaining

Lemma 144
If uniform hashing is used and collisions are resolved by chaining then an unsuccessful search runs in expected time $\Theta(1 + \alpha)$.

**Proof:** Any new key $k$ is equally likely to hash to any of the $m$ slots. The expected time to search unsuccessfully for $k$ in the list of $T[h(k)]$ is the time needed to search the list to its end, which has expected length $\alpha$.

- The situation for a successful search is slightly different because each list is not equally likely to be searched: If all stored KVPs are assumed to be equally likely to be retrieved then the probability that a list is searched is proportional to the number of KVPs which it contains.
- Still, one can prove that a successful search can be expected to involve $1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$ items.

Theorem 145
If uniform hashing is used and collisions are resolved by chaining then any search runs in expected time $\Theta(1 + \alpha)$.

- Obvious goal: Ensure that $\alpha = O(1)$. E.g., ensure $\alpha \leq 2$. 
Re-Hashing

- Assume that repeated insertions caused the load factor $\alpha$ to get too big.
- Then it is common to double $m$, thus getting a new $m^\star \approx 2m$ as new size of the hash table, implying a new load factor $\alpha^\star \approx \frac{1}{2} \alpha$:
  - Find a new hash function $h^\star : U \rightarrow \mathbb{Z}_{m^\star}$.
  - Re-hash: Insert each KVP from old hash table into new hash table.
- Same as for dynamic arrays, this adds $O(m^\star + n)$ time to one particular insertion, but happens rarely: It adds $O(1)$ amortized time to insertion.
- Re-hashing ensures that $\alpha \in O(1)$.

**Theorem 146**

If uniform hashing — with re-hashing as outlined above — is used and collisions are resolved by chaining then insertion, retrieval and deletion run in expected amortized time $O(1)$.

- But worst-case time is $\Theta(n)$ for a hash table with $n$ KVPs!
Choosing a “Good” Hash Function

- The analysis of hashing based on separate chaining relies on the assumption that the hash function satisfies the condition of uniform hashing.
- If the keys are random real numbers distributed independently and uniformly in the range \([0, 1]\) then

\[
h(k) := \lfloor k \cdot m \rfloor
\]

satisfies the condition of uniform hashing.
- However, in general this assumption is difficult to support in practice:
  - We will rarely know the probability distribution from which the keys are drawn.
  - Worse, the keys might not be drawn independently.
- Hence, heuristics are employed that tend to work well in practice.
Choosing a “Good” Hash Function: Modular Hashing

Modular hashing

\[ h(k) := k \mod m. \]

- Aka: Division method.
- We need to choose \( m \) carefully!
- If \( m \) is a power of 2, say \( m := 2^p \), then \( h(k) \) would amount to the \( p \) lowest-order bits of \( k \).
- This is a very poor choice for \( h \) unless we were guaranteed that all low-order \( p \)-bit patterns of the keys are equally likely.
- Similarly, if \( m := 2^p - 1 \) and \( k \) is a character string interpreted in radix \( 2^p \), then permuting the characters of \( k \) would not result in a different hash value: We have

\[
a \cdot (2^p)^i + b \cdot (2^p)^j \equiv_m a \cdot (1)^i + b \cdot (1)^j = a + b \equiv_m a \cdot (2^p)^j + b \cdot (2^p)^i.
\]

- Prime numbers not too close to a power of 2 (or power of 10) work well in practice as a choice for \( m \).
Choosing a “Good” Hash Function: Multiplication Method

**Multiplication method**

Let $x \in \mathbb{R}$ with $0 < x < 1$. Then

$$h(k) := \lfloor m \cdot (x \cdot k \mod 1) \rfloor,$$

where $x \cdot k \mod 1 := x \cdot k - \lfloor x \cdot k \rfloor$, i.e., $x \cdot k \mod 1$ is the fractional part of $x \cdot k$.

- This is a generalization of modular hashing: If $x := \frac{1}{m}$ then

  $$h(k) = \lfloor m \left( \left( \frac{1}{m} \cdot k \right) \mod 1 \right) \rfloor = \lfloor m \frac{k \mod m}{m} \rfloor = k \mod m.$$  

- [Knuth, TAoCP Vol. 3, 1973]: Supposedly $x := \frac{\sqrt{5}-1}{2}$ works well (“Fibonacci hash”).

- The multiplication method tends to yield hashes with decent “randomness” for the same reason why linear congruential generators work.

- The choice of $m$ is not so critical, and there seems to be some disagreement on what is best.
Choosing a “Good” Hash Function: Multiplication Method

- If \( m := 2^p \) and the word size of the machine is \( w \) bits, with \( p \leq w \), and if the key \( k \) fits into one word:
  1. Let \( x := \frac{s}{2^w} \) for some integer \( s \) in the range \( 0 < s < 2^w \).
  2. Multiply \( k \) by \( s \).
  3. This gives a \( 2w \)-bit value \( A_12^w + A_0 \), where \( A_1 \) is the high-order word of the product and \( A_0 \) is the low-order word of the product.
  4. Now take the \( p \) highest bits of \( A_0 \).

- This can be encoded compactly as \( s \cdot k \gg (w - p) \). (Use unsigned integers!)
- In order to meet Knuth’s suggestion, for \( w = 32 \) one could set \( s := 2654435769 \), since \( x := \frac{s}{2^{32}} = 0.618033988634124398231506347656250 \) is close to \( \sqrt{5} - 1 \).
- E.g., \( k := 123 \), \( p := 10 \), \( s := 2654435769 \) and \( w := 32 \). Then \( m = 2^{10} \) and

\[
y := k \cdot x \mod 1 = 0.01818060199730098247528076171875.
\]

Finally,

\[
\lfloor m \cdot y \rfloor = 18, \quad \text{i.e., } h(k) = 18.
\]

On the other hand,

\[
k \cdot s = 326495599587 = 76 \cdot 2^{32} + 78085091,
\]

and since \( 78085091 = (000001001010011101110111100011)_2 \), we get

\[
(0000010010)_2 = 18.
\]
Universal Hashing

- For some fixed hash function, a malicious adversary can always choose $n$ keys which all hash to the same slot, yielding an average retrieval time of $\Theta(n)$.

**Universal hashing, Dt.: Universelles Hashing**

The hash function is chosen randomly (from a diligently designed class of hash functions) in a way which is independent of the keys that will be stored.

- Pro: As for randomized quicksort, randomization guarantees that no sequence of inputs/operations will always result in a worst-case performance.
- Con: Universal hashing may (and, likely, will) behave differently for each execution, even when supplied with the same sequence of inputs/operations.

**Definition 147 (Universal collection of hash functions)**

Let $m \in \mathbb{N}$ and $\mathcal{H}$ be a finite collection of hash functions that map a universe $U$ of keys to $\{0, 1, \ldots, m - 1\}$. This collection of hash functions is *universal* if

$$|\{h \in \mathcal{H} : h(k) = h(i)\}| \leq \frac{|\mathcal{H}|}{m}$$

for each pair of distinct keys $k, i \in U$. 
Universal Hashing

**Lemma 148**

Let $m \in \mathbb{N}$ and $\mathcal{H}$ be a universal collection of hash functions. Consider a pair of distinct keys $k, i \in U$ and pick a hash function $h$ randomly from $\mathcal{H}$. Then

$$\Pr(h(k) = h(i)) \leq \frac{1}{m}.$$

**Proof:** There are at most $\frac{|\mathcal{H}|}{m}$ hash functions with $h(k) = h(i)$, out of a total of $|\mathcal{H}|$ hash functions.

- Hence, the probability of a collision is exactly the same as when choosing $h(k)$ and $h(i)$ randomly and independently from $\mathbb{Z}_m$.
- We will now analyze the expected complexity of universal hashing that uses separate chaining to resolve collisions.
- Note: The expectations will be over the choice of the hash function! No assumption is made about the distribution of the keys.
Universal Hashing

**Theorem 149**

Let \( m \in \mathbb{N} \) and \( H \) be a universal collection of hash functions. Pick a hash function \( h \) randomly from \( H \) and suppose that it has been used to hash \( n \) keys into a hash table \( T \) of size \( m \), with separate chaining used to resolve collisions.

If the key \( k \) is not in \( T \) then

\[
E(n_{h(k)}) \leq \alpha.
\]

If the key \( k \) is in \( T \) then

\[
E(n_{h(k)}) \leq 1 + \alpha.
\]
Universal Hashing

**Theorem 150**

Let $T$ be an initially empty hash table with $m$ slots, with separate chaining used to resolve collisions. If universal hashing is used then any sequence of $N$ insert, retrieve and delete operations that contains $O(m)$ insert operations runs in $\Theta(N)$ expected time.

**Proof:** Since we have $O(m)$ inserts among a total of $N$ operations, we get $n = O(m)$ and, thus, $\alpha = O(1)$. Then Theorem 149 tells us that one search runs in expected time $O(1)$. Same for one insert or one delete (once position is known). By linearity of expectation, the expected time of the entire sequence is $O(N)$ and, thus, also $\Theta(N)$.

- What remains to be done is to design a universal collection of hash functions . . .
Universal Hashing

Let $p \in \mathbb{P}$ be a prime number large enough such that $U \subseteq \mathbb{Z}_p$ and such that $p > m$.

**Definition 151**

For $a \in \mathbb{Z}_p^+$ and $b \in \mathbb{Z}_p$, we define the hash function $h_{a,b,p,m} : \mathbb{Z}_p \to \mathbb{Z}_m$ as follows:

$$h_{a,b,p,m}(k) := ((a \cdot k + b) \mod p) \mod m.$$  

Then

$$\mathcal{H}_{p,m} := \{h_{a,b,p,m} : a \in \mathbb{Z}_p^+, b \in \mathbb{Z}_p\}.$$  

We have $|\mathcal{H}_{p,m}| = (p - 1) \cdot p$.

**Theorem 152**

The class $\mathcal{H}_{p,m}$ is a universal collection of hash functions.
Choosing a “Good” Hash Function: Strings as Keys

- Most hash functions assume that all keys belong to $\mathbb{N}_0$.
- Standard way to map a character string $s$ to an integer: Interpret the string as an integer expressed in a suitable radix notation.
- E.g., since $p \sim 112$, $t \sim 116$ and $r \sim 114$ in the 7-bit ASCII code, we can regard the string $s := \text{ptr}$ as the triple $(112, 116, 114)$.
- Expressed as a radix-$R$ integer, with radix $R := 128$ (or radix $R := 256$), we get the mapping
  \[
  f(s) = 128^2 \cdot 112 + 128 \cdot 116 + 114 = 1849970.
  \]
- Now use $f(s)$ as argument for the hash function.
- Note: $f(s)$ can be truly huge! Hence, apply modulo computations early and do not compute the powers of the radix explicitly.
Choosing a “Good” Hash Function: Strings as Keys

```c
int StringModularHash(string S, // string in ASCII
    int R,       // radix
    int M)       // modulus
{
    N = S.length - 1;
    int h = S[N];    // modular hash of S
    for (i = N-1; i >= 0; --i) {
        h *= R;
        h += S[i];
        h = h mod M;
    }

    return h;
}
```
Perfect Hashing

- **Static set of keys:** Assume that the set of keys does not change once stored in the hash table. (E.g., consider the set of reserved words for a programming language, or the names of streets of a map uploaded to a navigation system.)

- Goal: Improve the excellent average-case performance of (universal) hashing to an excellent worst-case performance.

Perfect Hashing [Fredman&Komlós&Szemerédi (1984)]

**Perfect hashing** is a two-level hash scheme, with universal hashing at each level. The secondary hashing is injective, thus guaranteeing $\Theta(1)$ search time for a set of keys known a priori.

- Let $p \in \mathbb{P}$ be a prime number large enough such that $U \subseteq \mathbb{Z}_p$ and $p > m$.

- We hash $n$ keys of $U$ into $m$ slots of $T$ using universal hashing with open chaining. This primary hash function belongs to $\mathcal{H}_{p,m}$ and is of the form

$$h_{a,b,p,m}(k) := ((a \cdot k + b) \mod p) \mod m,$$

with $a \in \mathbb{Z}_p^+$ and $b \in \mathbb{Z}_p$. 
Perfect Hashing

Lemma 153

If we store $n$ keys in a hash table of size $m := n^2$ by using a hash function randomly chosen from a universal collection of hash functions, then we get collisions with a probability of less than $\frac{1}{2}$.

- Hence, after trying a few randomly chosen hash functions, we will have found a hash function that does not yield collisions with very high probability: The probability that we have found a hash function without collisions after trying $i$ hash functions is at least $1 - \frac{1}{2^i}$.

- Let $Y$ be the random variable that represents the number of hash functions that need to be tried until no collision occurs. We have

$$E(Y) = \sum_{i=1}^{\infty} 1 \cdot \Pr(Y \geq i) \leq \sum_{i=1}^{\infty} \frac{1}{2^{i-1}} = \sum_{i=0}^{\infty} \frac{1}{2^i} = 2.$$ 

Hence, we can expect that two random tries of hash functions suffice.
Perfect Hashing

- Still, it is obvious that a hash table of size $n^2$ is excessively large.
- Therefore, we apply this idea in a second round of hashing: The $n_j$ keys hashed to a slot $j$ of $T$ are re-hashed by a secondary hash function into a secondary hash table $S_j$ of size $m_j := n_j^2$, using a hash function $h_j$ chosen from $\mathcal{H}_{p,m_j}$.
- For the primary hash table we use $m := n$, resulting in $O(n)$ memory being consumed by the primary hash table.
- Of course, besides ensuring that no collisions occur in $S_j$, the overall space requirement shall remain linear.

**Lemma 154**

We store $n$ keys in the primary hash table $T$ of size $m := n$, using a hash function randomly chosen from a universal collection of hash functions. Let $n_j$ be the number of keys hashed into slot $j$, and let $m_j := n_j^2$ be the size of the secondary hash table $S_j$. Then the expected amount of memory consumed by all secondary hash tables is less than $2n$. 
Perfect Hashing

Theorem 155

Perfect hashing allows to store a fixed set of $n$ keys in expected $O(n)$ time and space in a two-level hash table such that search queries can be answered in worst-case $O(1)$ time.

- **Dynamic perfect hashing**: The hash function is updated whenever the set of keys changes. Allowing updates makes the situation quite messy, though . . .
Cuckoo Hashing

Cuckoo Hashing [Pagh&Rodler 2001]

Cuckoo hashing is a variant of open addressing that uses two hash functions $h_1, h_2$ such that any key $k$ is always either at slot $h_1(k)$ or at $h_2(k)$.

- Hence, search and delete operations are trivial and run in $O(1)$ worst-case time.
- The two hash functions may address the same hash table or two different tables.

Theorem 156

Cuckoo hashing guarantees worst-case $O(1)$ search and delete times. If the load factor is kept less than $\frac{1}{2}$ then an insert runs in expected $O(1)$ time.

- The bound on the expected time of insertion is rather tricky to prove. We sketch only how insertions work.
- Experiments suggest that cuckoo hashing is much faster than chaining for small hash tables, and slightly worse than perfect hashing. But it is dynamic!
- Variation: Use three or more hash functions to allow to increase the load factor.
Cuckoo Hashing: Insertion

- If $T[h_1(k)]$ is empty, then insert $k$ at $T[h_1(k)]$.
- Else, if $T[h_2(k)]$ is empty then insert at $T[h_2(k)]$.
- If both $T[h_1(k)]$ and $T[h_2(k)]$ are full then
  1. “kick the key $k_1$ stored at $T[h_1(k)]$ out of the nest”,
  2. store $k$ at $T[h_1(k)]$,
  3. store $k_1$ at $T[h_2(k_1)]$; if $T[h_2(k_1)]$ is occupied by $k_2$ then “kick $k_2$ out of the nest”, alternating between $h_1$ and $h_2$, etc.
- To prevent a loop (or large number of iterations) we break after some number $i$ (that is logarithmic in $m$) of iterations and re-build the hash table with larger $m$.

$h_1(8) = 2, h_2(8) = 8$
$h_1(4) = 0, h_2(4) = 5$
$h_1(1) = 4, h_2(1) = 5$
$h_1(9) = 0, h_2(9) = 8$
$h_1(5) = 0, h_2(5) = 8$
$h_1(3) = 0, h_2(3) = 4$
Multi-Dimensional Data and Geometric Queries

- Range Searching
- kd-Tree
- Quadtree
- Geometric Hashing
Introduction to Geometric Searching

Point-Inclusion Query: In which “cell” (of, e.g., a map) does a query point lie?

Range Searching:
- **Report Query**: Which points are within a query object (rectangle, circle)?
- **Count Query**: Only the number of points within an object matters.

Another way to distinguish geometric searching queries:

- **Single-Shot Query**: Only one query per data set.
- **Repetitive-Mode Query**: Many queries per data set; preprocessing may make sense.

The complexity of a query is determined relative to four cost measures:
- query time,
- preprocessing time,
- memory consumption,
- update time (in the case of dynamic data sets).
Range Searching: Report Query

**Problem: RANGESEARCHREPORT**

**Input:** A set $S$ of $n$ points in $\mathbb{R}^k$ and a query (hyper-)rectangle $R$.

**Report:** Those points of $S$ which are within $R$.

- **Case $k = 1$:** Then the rectangle $R$ is an interval.
  - As preprocessing we sort the points. This needs $O(n \log n)$ time.
  - A query is solved by a binary search which needs $O(\log n + m)$ time, where $m$ is the (output-sensitive) number of points returned.

- **Case $k \geq 2$:** The goal is to “extend” binary search to higher dimensions.
Range Searching: kd-Tree

For points \( p_1, \ldots, p_n \) in \( \mathbb{R}^2 \) we build a kd-tree ("\( k \)-dimensional (binary search) tree") as preprocessing:

- We start by finding the median \( p_m \) of the points with respect to their \( x \)-coordinates. (W.l.o.g.: “general position assumed!”)
- The point \( p_m \) becomes the root of the tree; it is labeled “vertical”.
- We divide the plane by a vertical straight line through \( p_m \) into two half-planes.
Range Searching: kd-Tree

- For points \( p_1, \ldots, p_n \) in \( \mathbb{R}^2 \) we build a kd-tree as the preprocessing:
  - Within each half-plane we find the medians with respect to the \( y \)-coordinates of the respective points.
  - These two points are called “horizontal” nodes and become the left and the right child of the root.
  - The recursive subdivision, alternating between \( x \)- and \( y \)-coordinates, continues until all points form nodes of the tree.
Suppose that a query rectangle $\mathcal{R} := [x_1, x_2] \times [y_1, y_2]$ is given for $x_1, x_2, y_1, y_2 \in \mathbb{R}$ with $x_1 \leq x_2$ and $y_1 \leq y_2$. 
Complexity of Range Searching Based on a kd-Tree

**Theorem 157**

Range searching based on a kd-tree in two dimensions needs $O(n \log n)$ preprocessing time, with $O(n)$ space complexity. A query can be carried out in $O(\sqrt{n} + m)$ time, where $m$ is the number of nodes reported.

**Sketch of Proof:** We focus on the query complexity. Fix one supporting line $\ell$ of the query rectangle. W.l.o.g., $\ell$ is vertical. Let $Q^x(n)$ be the maximum number of nodes of the kd-tree which are discriminated relative to $\ell$ if the root of the kd-tree is split according to $x$-coordinate. Similarly for $Q^y(n)$. We get for $\ell$ being vertical:

$$Q^x(n) = 1 + Q^y\left(\frac{n}{2}\right) \quad \text{and} \quad Q^y(n) = 1 + 2Q^x\left(\frac{n}{2}\right).$$

Induction allows to show

$$Q^x(n) = 2 + 2Q^x\left(\frac{n}{4}\right).$$

The Master Theorem 31 tells us that $Q^x \in O(\sqrt{n})$ and, thus, also $Q^y \in O(\sqrt{n})$. (One could also prove directly $Q^x(n) \leq 3\sqrt{n} - 2$.) Hence, $O(\sqrt{n})$ nodes are discriminated relative to the four supporting lines of the query rectangle.
The complexity of building a kd-tree does not change if no pre-sorting is carried out and a linear-time algorithm for median finding is used for identifying the nodes of the tree. (It just makes the implementation more tedious.)

For a range search in $\mathbb{R}^k$ we split the space alternatingly by straight lines ($k = 2$), planes ($k = 3$), or hyper-planes (for $k \geq 4$).

More efficient methods are known. But kd-trees are a very versatile tool in low dimensions!

**Theorem 158**

Range searching based on a $k$-dimensional kd-tree needs $O(k \cdot n \log n)$ preprocessing time, with $O(n)$ space complexity. A query can be carried out in $O(k \cdot n^{1 - 1/k} + m)$ time, where $m$ is the number of nodes reported.

Note the curse of dimensionality for large values of $k$: We ought to have $n \gg 2^k$ in order to make kd-trees practical!
Consider a bounding box of the points $p_1, \ldots, p_n$ in $\mathbb{R}^2$. E.g., pick $m_1, m_2 \in \mathbb{N}_0$ and transfer the points such that they fit into the workspace $[0, 2^{m_1}] \times [0, 2^{m_2}]$.

We subdivide the (rectangular) workspace recursively into four sub-rectangles (“cells”) by bisecting it in both $x$ and $y$.

The recursion stops when either a cell contains at most one point or a maximum depth — i.e., minimum cell size — is reached.
Quadtree

- Range queries in such a (point) quadtree are very similar to queries in a kd-tree.
- Insertion and deletion of points are supported easily.
- Quadtrees (Dt.: Quaternärbaum) are simple and easy to implement and tend to be quite efficient in practice.
- However, even just three points can result in a quadtree of huge height!
- Higher dimensions: In $\mathbb{R}^d$ split into $2^d$ hyper-rectangles.
- Rarely used for $d > 3$. When $d = 3$: octree.

**Theorem 159**

Consider a quadtree on the distinct points $p_1, p_2, \ldots, p_n$ such that every cell is either empty or contains one point. Then its height is in $\Theta(\Delta)$, where

$$\Delta := \frac{\max_{1 \leq i < j \leq n} d(p_i, p_j)}{\min_{1 \leq i < j \leq n} d(p_i, p_j)}.$$
Quadtree

- Quadtrees and octrees are widely used for representing a shape approximately: 
  *region quadtree* and *region octree*.
Nearest Neighbor Search

**Problem: NEARESTNEIGHBORSEARCH**

**Input:** A set $S$ of $n$ points in the Euclidean plane.

**Output:** The point of $S$ which is closest to a query point $q$, for a given point $q$.

- We do already know that the worst-case complexity of \textsc{NearestNeighborSearch} for $n$ points has an $\Omega(\log n)$ lower bound.
- Easy to solve in $O(n)$ time per query.
- A worst-case optimum $O(\log n)$ query is possible, based on tools of computational geometry.
Geometric Hashing: Regular Rectangular Grid

- Let $S := \{p_1, p_2, \ldots, p_n\}$.
- The bounding box of $S$ (or of a larger region that contains $S$) is partitioned into rectangular cells of uniform size by means of a regular grid.
- For every cell $c$, all points of $S$ that lie in $c$ are stored in a list associated with $c$.
- That is, the cells of the grid become the slots of the standard hash table, and the hash function assigns a point $p \in S$ to $c$ if and only if $p$ lies within $c$.
- We can cover the entire plane by extending boundary cells to infinity.
Determine the cell $c$ in which the query point $q$ lies.
By searching in $c$ (and possibly in its neighboring cells, if $c$ is empty), we find a first candidate for the nearest neighbor.
Let $\delta$ be the distance from $q$ to this point.
We continue searching in $c$ and in those cells around $c$ which are intersected by a disk $D$ with radius $\delta$ centered at $q$.
Whenever a point of $S$ is found that is closer to $q$ than $\delta$, we reduce $\delta$ appropriately.
The search stops once no unsearched cell exists that is intersected by the disk $D$. 
Geometric Hashing: Grid Resolution

- What is a suitable resolution of the grid? There is no universally valid answer. In any case, the grid should not use more than $O(n)$ memory!

**Personal experience**

- Grids of the form $(w \cdot \sqrt{n}) \times (h \cdot \sqrt{n})$ seem to work nicely, with $w \cdot h = c$ for some constant $c$.
- The parameters $w, h$ are chosen to adapt the resolution of the grid to the aspect ratio of the bounding box of the points.
- By experiment: $1 \leq c \leq 2$.

- This basic scheme can be tuned considerably:
  - Switch to multi-level hashing or to kd-trees if a small sample of the points indicates that the points are distributed highly non-uniformly.
  - Adapt the grid resolution and re-hash if the number of points stored changes significantly due to insertions and deletions of points.
  - Hash-based nearest-neighbor searching will work best for points that are distributed uniformly, and will fail miserably if all points end up in one cell!
  - Still, personal experience tells me that (tuned) geometric hashing works extremely well even for point sets that are distributed highly irregularly!
Geometric Hashing and Range Searching

- Geometric hashing can also be used to answer (generalized) range queries quite efficiently.
- E.g., one may need to report those points of $S$ that lie within a query triangle.
- Then it suffices to check those points of $S$ which are stored in cells overlapped by (the bounding box of) the triangle.
Hard Problems and Approximation Algorithms

- Intractability
- P and NP
- NP-Hard and NP-Complete
- Proving NP-Completeness
- Approximation Algorithms
- Problems of Unknown Complexity
Tractable vs. Intractable Problems

Definition 160 (Polynomially solvable, Dt.: in polynomialer Zeit lösbar)

A problem $P$ is solvable in polynomial time if there exists a polynomial $p$ such that a solution for every instance of $P$ can be obtained in time $O(p)$, where the size of the instance/input forms the argument of the polynomial.

Note: $O(n)$, $O(n \log^2 n)$, $O(2^{1000} n^3 \sqrt{n})$ and $O(n^{1000})$ all are polynomial bounds.

A polynomial-time algorithm need not be practical: Even an $O(n^2)$ algorithm might already be impractical on realistic sizes of problems!

Motivation for distinguishing between polynomial and non-polynomial problems:

- If a problem is not solvable in polynomial time then there is absolutely no hope for an efficient (exact) solution for absolutely all large inputs. Such problems are considered intractable.
- Polynomials have a nice closure property under standard operations: If $p$ and $q$ are polynomials then $p \circ q$ is a polynomial for most “standard” operations $\circ$.
- If a problem of size $n$ is solvable in time $f(n)$ on one model of computation then it is also solvable in $f(p(n))$ on another model of computation, for virtually all “natural” models of computation and suitable polynomial $p$. 
Sample Non-Polynomial Problem: Towers of Hanoi

- Tower-of-Hanoi Problem (ToH): Given three poles (labeled I, II, III) and a stack of \( n \) disks arranged on Pole I from largest at the bottom to smallest at the top, we are to move all disks to Pole II such that only one disk is moved at a time and such that no larger disk ever is placed on a smaller disk.

- Attributed to Édouard Lucas [1883]. Supposedly based on an Indian legend about Brahmin priests moving 64 disks in the Great Temple of Benares; once they are finished, life on Earth will end.

- Goal: Find an algorithm that uses the minimum number of moves.

One can prove: A (straightforward) recursive algorithm needs \( 2^n - 1 \) moves.

One can also prove: Every(!) algorithm that solves ToH needs at least \( 2^n - 1 \) moves.

Thus, the solution achieved by the recursive algorithm is optimal as far as the number of moves is concerned: No polynomial-time solution exists!

Note, though, that there exists a simple iterative solution due to Buneman&Levy [1980] which avoids an exponential-sized stack!
Definition 161 (Problem Class $\mathcal{P}$)

The problem class $\mathcal{P}$ is the class of all decision problems that are solveable in polynomial time by a deterministic algorithm.

- Intuitively, the class $\mathcal{NP}$ is the class of those decision problems for which one can verify on a deterministic computer in polynomial time whether or not an alleged solution (“certificate”) is indeed a correct solution that allows the algorithm to answer “yes”.

- For instance, while it appears difficult to find assign colors to nodes of a graph such that the minimum amount of colors is used, it is easy to check (in polynomial time) whether a suggested assignment of colors yields a proper coloring.
Often, \( \mathcal{NP} \) is informally described as the class of decision problems that can be solved by a non-deterministic algorithm in polynomial time.

Roughly, a non-deterministic algorithm consists of a “guessing” phase and a “verifying phase”.

1. **Guessing**: An arbitrary string \( s \) of characters is generated.
2. **Verifying**: A deterministic algorithm takes the input and the string \( s \). It may use or ignore \( s \) during its computation. Eventually, it returns the correct answer “yes” or “no”, or it may get in an infinite loop and never halt.

The time consumed by a non-deterministic algorithm is the time needed to write \( s \) plus the time consumed by the deterministic verifying phase.

**Definition 162 (Non-deterministic Polynomial-time Solution)**

A *non-deterministic algorithm solves a decision problem \( P \) in polynomial time* if there is a fixed polynomial \( p \) such that for every instance \( x \) of \( P \) for which the answer is “yes” there is at least one execution of the algorithm that returns “yes” in at most \( p(|x|) \) time.
\[ \mathcal{P} \text{ and } \mathcal{NP} \]

**Definition 163 (Problem Class \( \mathcal{NP} \))**

The problem class \( \mathcal{NP} \) is the class of all decision problems that are solveable in polynomial time by a non-deterministic algorithm.

- Note that being in \( \mathcal{NP} \) does not imply for a problem that we can easily find a certificate \( s \) for an instance if the answer is “yes”. But there has to exist a polynomial-time algorithm to check the validity of a proposed certificate \( s \).

**Lemma 164**

We have \( \mathcal{P} \subseteq \mathcal{NP} \).

**Sketch of Proof:** Let \( P \in \mathcal{P} \). All we need to do is to apply a deterministic algorithm that solves \( P \) in polynomial time and let it ignore any non-determinism.

- It is widely believed that \( \mathcal{P} \neq \mathcal{NP} \) . . .
- Want to become rich and famous? In 2000, the Clay Mathematics Institute (CMI) at Cambridge, Massachusetts (USA), named seven Millennium Prize Problems and designated a $7 million prize fund for the solution of these problems, with $1 million allocated to each problem. And the \( \mathcal{P} = \mathcal{NP} \) question is one of them!
Polynomial Reducibility

**Definition 165 (Polynomially Reducible, Dt.: polynomial reduzierbar)**

A decision problem $P$ is *polynomially reducible* (or simply *reducible*) to a decision problem $Q$, denoted by $P \leq_p Q$, if there exists a reduction from $P$ to $Q$ that runs in polynomial time.

- This definition can easily be extended to cover reductions from a decision problem $P$ to an arbitrary problem $Q$ by requesting that the output generated by an instance of $Q$ allows to decide in polynomial time whether the answer for the original instance of $P$ is "yes" or "no".
- Such an extension allows a reduction from a combinatorial decision problem to a combinatorial optimization problem.

**Lemma 166**

For two decision problems $P, Q$, if $P \leq_p Q$ and $Q \in \mathcal{P}$ then $P \in \mathcal{P}$.

**Proof**: Let $x$ be an instance of $P$. We apply a polynomial reduction and map $x$ in time $p(|x|)$ to an instance $t(x)$ of $Q$. Since $Q \in \mathcal{P}$, a solution to $t(x)$ can be obtained in time $q(p(|x|))$, where $q(|\alpha|)$ denotes the time needed for solving an instance $\alpha$ of $Q$. 
\( \text{NP}\)-Hardness and \( \text{NP}\)-Completeness

**Definition 167 (\( \text{NP}\)-hard, Dt.: \( \text{NP}\)-schwer)**

A problem \( Q \) is \( \text{NP}\)-hard if every problem \( P \in \text{NP} \) is polynomially reducible to \( Q \).

- Note that \( \text{NP}\)-hard does not mean “\( \text{NP}\) and hard”! Rather, it means “at least as hard as any problem in \( \text{NP}\)”. 

**Definition 168 (\( \text{NP}\)-complete, Dt.: \( \text{NP}\)-vollständig)**

A problem \( Q \) is \( \text{NP}\)-complete if it is in \( \text{NP} \) and if it is \( \text{NP}\)-hard. The class of \( \text{NP}\)-complete problems is denoted by \( \text{NPC} \).

- Hence, an optimization problem might be \( \text{NP}\)-hard, but only decision problems can be \( \text{NP}\)-complete.

**Lemma 169**

We have \( \text{NPC} \subseteq \text{NP} \) and \( \text{NPC} \subset \text{NP}\)-hard.
NP-Hardness and NP-Completeness

**Theorem 170**

If some \( \mathcal{NP} \)-complete problem is in \( \mathcal{P} \) then \( \mathcal{P} = \mathcal{NP} \).

**Proof:** If an \( \mathcal{NP} \)-complete problem \( P \) is in \( \mathcal{P} \), then all problems of \( \mathcal{NP} \) can be reduced in polynomial time to \( P \) and, thus, also solved in polynomial time.

**Theorem 171 (Cook 1971)**

The satisfiability problem of propositional logic, \( \text{SAT} \), is \( \mathcal{NP} \)-complete.

- Dt.: Erfüllbarkeitsproblem der Aussagenlogik.
- Which other problems are \( \mathcal{NP} \)-complete?
- In 1972, Karp established the \( \mathcal{NP} \)-completeness of 21 combinatorial and graph-theoretical computational problems.
- See Garey and Johnson, "Computers and Intractability: A Guide to the Theory of \( \mathcal{NP} \)-Completeness". (This used to be the bible of \( \mathcal{NP} \)-completeness.)
- In the meantime, a few thousand problems are known to be \( \mathcal{NP} \)-complete . . .
### A List of $\mathcal{NP}$-Complete Problems

<table>
<thead>
<tr>
<th>Problem: SAT-CNF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A propositional formula $A$ which is in conjunctive normal form.</td>
</tr>
<tr>
<td><strong>Decide:</strong> Is $A$ satisfiable?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem: 3-SAT-CNF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A propositional formula $A$ which is in conjunctive normal form such that every clause consists of exactly (or at most) three literals.</td>
</tr>
<tr>
<td><strong>Decide:</strong> Is $A$ satisfiable?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem: SUBSETSUM, DT.: TEILSUMMENPROBLEM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A set $S$ of $n$ natural numbers and a number $m \in \mathbb{N}$.</td>
</tr>
<tr>
<td><strong>Decide:</strong> Does a subset of the numbers of $S$ add up to exactly $m$?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem: BINPACKING, DT.: BEHÄLTERPROBLEM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A set $S$ of $n$ objects with sizes $s_1, s_2, \ldots, s_n \in \mathbb{Q}$, where $0 &lt; s_i \leq 1$, and a number $k \in \mathbb{N}$.</td>
</tr>
<tr>
<td><strong>Decide:</strong> Do the objects fit into $k$ bins of unit capacity?</td>
</tr>
</tbody>
</table>
A List of $NP$-Complete Problems

**Note**

It is common not to make an explicit distinction between a decision problem, as listed, and its optimization variant (if it exists): For the optimization problem we drop “and a number $k$” and replace “decide” by “maximize $k$” or “minimize $k$”. (We will also be liberal in using the same name both for a decision problem and for its optimization variant . . .)

**Problem: Knapsack (Knap), Dt.: Rucksackproblem**

**Input:** A knapsack of capacity $c \in \mathbb{N}$ and $n$ objects with sizes $s_1, s_2, \ldots, s_n$ and “profits” $p_1, p_2, \ldots, p_n$. In addition, we are given a number $k \in \mathbb{N}$.

**Decide:** Is there a subset of the objects that fits into the knapsack and achieves a total profit of at least $k$?

**Problem: SetCover (SC), Dt.: Mengenüberdeckungsproblem**

**Input:** A set $S$ and a family $S := \{S_1, S_2, \ldots, S_m\}$ of $m$ subsets of $S$, for $m \in \mathbb{N}$, and a natural number $k \in \mathbb{N}$.

**Decide:** Do there exist at most $k$ subsets $S_{i_1}, S_{i_2}, \ldots, S_{i_k} \in S$ such that $S = S_{i_1} \cup S_{i_2} \cup \ldots \cup S_{i_k}$?
A List of $\mathcal{NP}$-Complete Problems

**Problem: HAMILTONIAN CYCLE (HC)**

**Input:** An undirected graph $\mathcal{G}$.

**Decide:** Does $\mathcal{G}$ contain a Hamiltonian cycle?

**Problem: HAMILTONIAN PATH (HP)**

**Input:** An undirected graph $\mathcal{G}$.

**Decide:** Does $\mathcal{G}$ contain a Hamiltonian path?

**Problem: TRAVELING SALESMAN PROBLEM (TSP), Dt.: RUNDREISEPROBLEM**

**Input:** A weighted and undirected graph $\mathcal{G}$, and a number $c \in \mathbb{R}^+$.

**Decide:** Does $\mathcal{G}$ contain a Hamiltonian cycle whose total cost is less than $c$?

**Problem: MINIMUM STEINER TREE (MST), Dt.: STEINERBAUMPROBLEM**

**Input:** A weighted and undirected graph $\mathcal{G} = (V, E)$, a set of required nodes ("terminals") $T \subset V$, and a number $c \in \mathbb{R}^+$.

**Decide:** Does there exist a connected subgraph $(V', E')$ of $\mathcal{G}$ such that $T \subseteq V'$ and the sum of the costs of the edges of $E'$ is less than $c$?
A List of \textit{NP}-Complete Problems

Problem: \textsc{VertexCover (VC)}, Dt.: \textsc{Knotenüberdeckungsproblem}

\begin{itemize}
  \item \textbf{Input:} An undirected graph $\mathcal{G} = (V, E)$ and a number $k \in \mathbb{N}$.
  \item \textbf{Decide:} Does there exist a vertex cover that has $k$ vertices? (A subset $C \subseteq V$ of the vertices of a graph $\mathcal{G}$ forms a vertex cover of $\mathcal{G}$ if every edge of $E$ is incident upon at least one vertex of $C$.)
\end{itemize}

\begin{itemize}
  \item \textsc{Vertex Cover}
  \item \textsc{Clique}
\end{itemize}

Problem: \textsc{Clique (CLIQ)}, Dt.: \textsc{Cliquenproblem}

\begin{itemize}
  \item \textbf{Input:} An undirected graph $\mathcal{G} = (V, E)$ and a number $k \in \mathbb{N}$.
  \item \textbf{Decide:} Does $\mathcal{G}$ have a clique of size $k$? (A subset $Q \subseteq V$ of the vertices of a graph $\mathcal{G}$ forms a clique of $\mathcal{G}$ if every pair of distinct vertices of $Q$ is linked by an edge of $E$.)
\end{itemize}

\begin{itemize}
  \item \textsc{Clique}
\end{itemize}
A List of \(NP\)-Complete Problems

**Problem: INDEPENDENTSET (IS), Dt.: STABILITÄTSPROBLEM**

**Input:** An undirected graph \(G = (V, E)\) and a number \(k \in \mathbb{N}\).

**Decide:** Does \(G\) have an independent set of size \(k\)? (A subset \(I \subseteq V\) of the vertices of a graph \(G\) forms an independent set if no pair of vertices of \(I\) is connected by an edge of \(E\)).

*Vertex Cover*

*Clique*

*Independent Set*

**Problem: \(k\)-COLORING (\(k\)-COL), Dt.: \(k\)-FÄRBBARKEIT**

**Input:** An undirected graph \(G = (V, E)\), and an integer \(k \in \mathbb{N}\).

**Decide:** Does \(G\) admit a coloring that uses at most \(k\) colors? (An assignment of colors to all vertices of \(V\) is called a (vertex) coloring if adjacent vertices are assigned different colors.)
Intuitively, ETSP ought to be \( \mathcal{NP} \)-complete, too.

Indeed, the \( \mathcal{NP} \)-completeness of ETSP is claimed in several publications . . .

However, this claim is wrong! (The title of [Papadimitriou 1977] is misleading!)

ETSP, and several other optimization problems involving Euclidean distance, are not known to be in \( \mathcal{NP} \) due to a “technical twist”: For ETSP, the length of a tour on \( n \) points (with integer/rational coordinates) is a sum of \( n \) square roots. Comparing this sum to a number \( c \) may require very high precision, and no polynomial-time algorithm is known for solving this problem. (E.g., repeated squaring of \( n \) square roots may lead to numbers that need \( O(2^n) \) bits to store.)

**Open problem**

Can the sum of \( n \) square roots of integers be compared to another integer in polynomial time?

**ETSP is \( \mathcal{NP} \)-hard . . .**

. . . but not known to be in \( \mathcal{NP} \).
The following decision problems are \( \mathcal{NP} \)-complete:

- \textsc{Sat-CNF},
- \textsc{3-Sat-CNF},
- \textsc{SubsetSum},
- \textsc{BinPacking},
- \textsc{Knapsack},
- \textsc{SetCover},
- \textsc{HamiltonianCycle},
- \textsc{HamiltonianPath},
- \textsc{TSP},
- \textsc{SteinerMinimumTree},
- \textsc{VertexCover},
- \textsc{Clique},
- \textsc{IndependentSet},
- \( k \)-\textsc{Col}. 

A List of $\mathcal{NP}$-Complete Problems in the Sciences

- $\mathcal{NP}$-completeness is not just a concern to theoreticians!
- Rather, many fundamental problems in the sciences have been shown to be $\mathcal{NP}$-complete (or $\mathcal{NP}$-hard).
- The following list was taken from “The status of the P versus NP problem” [Fortnow, CACM 2009]:
  - Finding a DNA sequence that best fits a collection of fragments of the sequence [Gusfield 1997].
  - Finding a ground state in the Ising model of phase transitions [Cipra 2000].
  - Finding Nash Equilibriums with specific properties in a number of environments [Conitzer 2008].
  - Finding optimal protein threading procedures [Lathrop 1994].
What $\mathcal{NP}$-Complete Does Not Imply

- While all known $\mathcal{NP}$-complete problems are indeed tremendously difficult to solve, solving an $\mathcal{NP}$-complete problem does not “necessarily” require exponential time: Otherwise, we would have $\mathcal{P} \neq \mathcal{NP}$!
- $\mathcal{NP}$-completeness does not imply that absolutely all (or even just most) instances of a problem are difficult. E.g., powerful SAT-solvers are known.
- $\mathcal{NP}$-complete problems are not the “most difficult” problems: They have a running time that is “only” exponential . . .
- [Presburger (1929)] introduced a first-order theory of the natural numbers with addition and equality, but without multiplication. Its axioms include some form of induction. For every sentence in Presburger arithmetic one can decide, i.e., determine algorithmically, whether it follows from the axioms of Presburger arithmetic.
- [Fischer&Rabin (1974):] The decision algorithm for *Presburger arithmetic* for a sentence of length $n$ has a worst-case running time of at least $2^{cn}$, for some constant $c > 0$. 

Proving \( \mathcal{NP} \)-Completeness of a Problem

**Theorem 173**

If \( P \leq_p Q \) and \( P \) is \( \mathcal{NP} \)-complete and \( Q \in \mathcal{NP} \) then \( Q \) also is \( \mathcal{NP} \)-complete.

**Proof:** Let \( R \) be in \( \mathcal{NP} \). We reduce an instance \( x \) of \( R \) to an instance \( t_1(x) \) of \( P \), and reduce \( t_1(x) \) to an instance \( t_2(t_1(x)) \) of \( Q \). This reduction runs in polynomial time. Hence, every problem that is in \( \mathcal{NP} \) can be reduced polynomially to \( Q \).

**Steps to prove a problem** \( Q \) **to be** \( \mathcal{NP} \)-complete

1. Show that \( Q \in \mathcal{NP} \).
2. Pick a problem \( P \) that is known to be \( \mathcal{NP} \)-complete.
3. Construct (or prove the existence of) a polynomial reduction from \( P \) to \( Q \).

\( \mathcal{NP} \)-completeness proofs tend to make extensive use of “gadgets”. The (fairly creative) process of designing such gadgets is sometimes called “gadgeteering.”

**Trevisan et al. (2000)**

A *gadget* is a finite combinatorial structure which translates a given constraint of one (optimization) problem into a set of constraints of a second (optimization) problem.
Sample \( \mathcal{NP} \)-Completeness Proof: \( 4\text{-COL} \)

**Lemma 174**

If \( 3\text{-COL} \) is \( \mathcal{NP} \)-complete then \( 4\text{-COL} \) is \( \mathcal{NP} \)-complete.

**Proof:**

- Suppose that \( 3\text{-COL} \) is \( \mathcal{NP} \)-complete. We show that \( 3\text{-COL} \) \( \leq_p \) \( 4\text{-COL} \). (Clearly, \( 4\text{-COL} \) is in \( \mathcal{NP} \).)
- Consider a graph \( G = (V, E) \). We transform \( G \) into a graph \( G' = (V', E') \) by adding a vertex \( v \notin V \) to \( V \). Also, we add edges from \( v \) to all nodes of \( V \). That is,
  - \( V' := V \cup \{v\} \), and
  - \( E' := E \cup \{uv : u \in V\} \).
- This transformation can be carried out in time polynomial in the number of nodes and edges of \( G \).
- Since \( v \) consumes one color which cannot be used for any other node of \( G' \), the graph \( G \) is 3-colorable exactly if \( G' \) is 4-colorable.

**Corollary 175**

If \( k\text{-COL} \) is \( \mathcal{NP} \)-complete for some \( k \in \mathbb{N} \) then \( (k + 1)\text{-COL} \) is \( \mathcal{NP} \)-complete.
Lemma 176

3-COL is \( \mathcal{NP} \)-complete.

Proof:

- Clearly, 3-COL is in \( \mathcal{NP} \). We prove 3-SAT-CNF \( \leq_p \) 3-COL. Given a 3-CNF expression \( e \), where every clause consists of exactly three literals, we show how to construct a graph \( G \) in polynomial time such that \( e \) is satisfiable if and only if \( G \) can be colored with three colors.
- Let \( k \) denote the number of clauses of \( e \). The \( n \) variables appearing in \( e \) are denoted by \( v_1, v_2, \ldots, v_n \).
- Hence, \( e \) contains at least one of the two literals \( v_i \) and \( \overline{v_i} \), for all \( i \in \{1, 2, \ldots, n\} \).
- We build an appropriate graph \( G \) that contains \( 2n + 6k + 3 \) nodes and \( 3n + 12k + 3 \) edges. This graph consists of
  - a graph representation of the variables, denoted by \( G_V \),
  - a graph representation of all clauses, \( G_C \), and of
  - appropriate edges to link \( G_V \) and \( G_C \) together.
Proof of Lem. 176 (cont’d):

- Construction of $G_V$ (to represent the variables):
  - Three special nodes — denoted by $C$ (for “control”), $T$ (for “true”), and $F$ (for “false”) — are linked into a triangle, the so-called control triangle.
  - For each variable $v$ we create two nodes — the “literal nodes” $v$ and $\bar{v}$ — and link them with the node $C$ and with each other to form a triangle.
  - This gives $2n + 3$ nodes and $3n + 3$ edges constructed so far for $G_V$. 

Sample $\mathcal{NP}$-Completeness Proof: 3-COL

Proof of Lem. 176 (cont’d):

- Clearly, three colors are necessary and sufficient to color $G_V$.
- Due to the use of the control node $C$, the variable nodes $v_i$ and $\bar{v}_i$ have to use the same colors as the nodes $T$ and $F$.
- If the colors of $v_i$ and $T$ match, then the colors of $\bar{v}_i$ and $F$ have to match, too.
- Intuitively, think of assigning the variable $v_i$ the value true if its node is colored with the same color as $T$. Similarly, coloring $v_i$ with the same color as $F$ can be interpreted as assigning the value false to $v_i$, thus, assigning the value true to $\bar{v}_i$. 

![Diagram of the proof](image.png)
Sample \(\mathcal{NP}\)-Completeness Proof: 3-COL

**Proof of Lem. 176 (cont’d):**

- Construction of \(G_C\) (to represent the clauses):
  - We use a clause gadget as depicted below, with one gadget per clause.
  - Each clause gadget is linked to five other nodes of \(G_V\):
    1. It is linked to the nodes \(C\) and \(T\) of the control triangle, and
    2. to three literal nodes corresponding to the literals that appear in the specific clause represented by the clause gadget.
  - The graph \(G_C\) is formed by \(k\) copies of this gadget, with one gadget per clause, resulting in a total of \(6k\) additional nodes and \(12k\) additional edges.

![Diagram](attachment:image.png)
Sample $\mathcal{NP}$-Completeness Proof: 3-Col

Proof of Lem. 176 (cont’d):

- The final graph $G$ consists of $G_V$ plus $G_C$, i.e., of $2n + 6k + 3$ nodes and $3n + 12k + 3$ edges. Clearly, $G$ can be constructed in time polynomial in the number of variables and clauses of $e$.
- Let $a, b, c$ be the literal nodes that are pointed at by the three edges of a clause gadget marked by “to Literal . . .”.
- Since $a, b, c$ are linked to $C$, the only colors feasible for $a, b, c$ are the two colors used for $T$ and $F$.
- A simple enumeration of all possible color assignments to $a, b, c$ shows that a clause gadget can be colored with three colors if and only if at least one of $a, b, c$ is colored with the same color as $T$. 

![Diagram](null)
Sample $NP$-Completeness Proof: 3-COL

Proof of Lem. 176 (cont’d):

- We conclude that $G$ can be colored with three colors exactly if there exists a consistent color assignment to all literal nodes such that at least one literal node of each clause is colored with the same color as $T$.
- Thus, the Boolean expression $e$ is satisfiable if and only if $G$ can be colored with three colors. 

\[\square\]
Sample \(\mathcal{NP}\)-Completeness Proof: Hamiltonian Triangulation

**Definition 177 (Hamiltonian triangulation)**

A triangulation (of points or of polygonal figures) is *Hamiltonian* if its dual graph admits a Hamiltonian cycle.

**Theorem 178 (Arkin et al. 1996)**

Testing whether a given simple polygon has a Hamiltonian triangulation can be done in \(O(|E|)\) time, where \(|E|\) is the number of visibility graph edges in the polygon.

**Theorem 179 (Arkin et al. 1996)**

Given a simple polygon with (simple polygonal) holes, it is \(\mathcal{NP}\)-complete to determine whether there exists a Hamiltonian triangulation of its interior.
Sample \(\mathcal{NP}\)-Completeness Proof: Hamiltonian Triangulation

Proof of Thm. 179:

- We prove this theorem by reducing the known \(\mathcal{NP}\)-complete problem of determining whether a planar cubic graph is Hamiltonian to it. (Obviously, deciding whether such a triangulation exists is in \(\mathcal{NP}\).)
- Given a straight-line plane drawing of a planar cubic graph \(G\), we construct a polygon with holes, where the holes correspond to the bounded faces of \(G\):
  - Each arc of \(G\) is mapped to a narrow “V”-shaped tunnel.
  - Thus, a node \(abc\) of \(G\) corresponds to three node-vertices \(a, b, c\) of the polygonal area.
  - Each arc of \(G\) introduces two arc-vertices, \(g, h\), of the polygonal area.
Sample $\mathcal{NP}$-Completeness Proof: Hamiltonian Triangulation

Proof of Thm. 179 (cont’d):

- We can construct the “V”-shaped tunnels such that the following properties hold:
  - The resulting polygons are simple and bound a polygonal area $P$ with $k$ holes if $G$ contained $k$ bounded faces.
  - A node-vertex is visible by another node vertex exactly if both correspond to the same node of $G$.
  - Every arc-vertex sees exactly its corresponding arc-vertex and the corresponding six node-vertices.
  - Pairs of arc-vertices form *forced diagonals* contained in every triangulation.
- This construction can be carried out in polynomial time.
Sample $\mathcal{NP}$-Completeness Proof: Hamiltonian Triangulation

Proof of Thm. 179 (cont’d):

- Suppose that $G$ admits a Hamiltonian cycle.
- One can show that there exists a triangulation of $P$ that is Hamiltonian.
- Now suppose that $P$ has a triangulation that is Hamiltonian.
- One can show that $G$ contains a Hamiltonian cycle.
- Recall that every triangulation contains the forced diagonals defined by the arc-vertices, which can be crossed by a Hamiltonian cycle at most once.
- Hence, the arcs of $G$ that correspond to forced diagonals that the cycle crosses once in the triangulation of $P$ form a Hamiltonian cycle in $G$.
- Summarizing, $G$ contains a Hamiltonian cycle if and only if $P$ admits a Hamiltonian triangulation.
Dealing with $\mathcal{NP}$-Hard Problems

- Many combinatorial optimization problems could be solved by a brute-force enumeration of all possibilities.
- E.g., we could solve ETSP for $n$ cities by enumerating all $(n - 1)!$ possible tours.
- According to legend, the power of exponential growth was already known by the Brahmin Sissa ibn Dahir (ca. 300-400 AD): As a reward for the invention of the game of chess (or its Indian predecessor Chaturanga) he asked his king, Shihram, to place one grain of rice in the first square of a chessboard, two in the second, four in the third, and so on, doubling the amount of rice up to the 64-th square. Needless to say, the king could not fulfill Sissa’s request . . .
- In short terms: An algorithm whose running time is $2^n$ or worse is all but useless for most practical applications!!
- Unfortunately, while proving a problem to be $\mathcal{NP}$-hard/complete might constitute quite an achievement, it tends to shed little light on how to solve it.
- So, what shall we do next?
- In the sequel, we will study algorithms that provide an approximation of the solution sought.
- But we will not just dive into heuristics: Our approximations will come with some guarantee of how far off they may be from the true solutions!
**Definition 180 (Approximation with guaranteed quality)**

For an instance of an optimization problem, let $APX > 0$ denote the numerical quantity achieved by an algorithm $\mathcal{A}$ that solves it approximately, and let $OPT > 0$ denote the true optimum. Let $p: \mathbb{N} \rightarrow \mathbb{R}^+$. Then the approximation $\mathcal{A}$ has quality $p$ if

$$\max \left\{ \frac{APX}{OPT}, \frac{OPT}{APX} \right\} \leq p(n)$$

holds for all input instances of size $n$.

- Of course, $p(n) \geq 1$ for all $n \in \mathbb{N}$.

**Note: $OPT$ is unknown!**

If we want to argue that some approximation algorithm has a particular guaranteed quality then we need to do so without knowing $OPT$!

**Definition 181 (Constant-factor approximation)**

An approximation algorithm with quality $p: \mathbb{N} \rightarrow \mathbb{R}^+$ is a constant-factor approximation with approximation factor $c \in \mathbb{R}^+$ if $p(n) \leq c$ holds for all (sufficiently large) $n \in \mathbb{N}$.
Polynomial-Time Approximation Scheme

Definition 182 (Polynomial-time approximation scheme (PTAS))

A polynomial-time approximation scheme (PTAS) for an optimization problem is an algorithm which takes as additional input a parameter $\varepsilon \in \mathbb{R}^+$ and generates a $(1 + \varepsilon)$-approximation for every instance of the optimization problem such that its running time is a polynomial in $n$ for problem instances of size $n$, for every fixed value of $\varepsilon$.

- Dt.: Polynomialzeitapproximationsschema.
- Common to PTAS algorithms is the fact that $O(1/\varepsilon)$ is allowed to appear as exponent of $n$ or $\log n$.
- We may even see complexity terms of the form $O(n^{\lceil 1/\varepsilon \rceil})$.
- A variant that is more useful in practice is a fully polynomial-time approximation scheme (FPTAS), for which we demand the time to be polynomial in both $n$ and $1/\varepsilon$.
- Quasi-polynomial-time approximation scheme (QPTAS): We get a complexity of $O(n^{\text{polylog} n})$ for every fixed $\varepsilon \in \mathbb{R}^+$. 
Approximate \textsc{SetCover}

\textbf{Theorem 183}

Consider a set $S$ with $n$ elements and a family $S := \{S_1, S_2, \ldots, S_m\}$ of $m$ subsets of $S$, with $\bigcup_{1 \leq i \leq m} S_i = S$. Then the following algorithm achieves an $(\ln n)$-approximation:

Repeat until all elements of $S$ are covered:

Pick the set $S_i$ with the largest number of uncovered elements.

\textbf{Proof:} Since it is obvious that the greedy algorithm achieves a cover, we focus on the approximation factor. Suppose that $k$ sets of $S$ suffice to cover $S$. Let $n_i$ be the number of elements of $S$ not yet covered after the $i$-th iteration of the algorithm, with $n_0 := n$.

Since $k$ sets suffice to cover also these remaining elements, one set (not yet picked by the algorithm) must exist in $S$ that contains at least $n_i/k$ of them. This implies

$$n_{i+1} \leq n_i - \frac{n_i}{k} = n_i \left(1 - \frac{1}{k}\right) \leq \ldots \leq n_0 \left(1 - \frac{1}{k}\right)^{i+1}.$$

The standard inequality $1 - x < e^{-x}$, for all $x \in \mathbb{R} \setminus \{0\}$, implies

$$n_i \leq n_0 \left(1 - \frac{1}{k}\right)^i < n_0 (e^{-1/k})^i = n \cdot e^{-i/k}.$$

Since $n \cdot e^{-\ln n} = 1$, we get $n_i < 1$ (and no uncovered elements) for $i := k \ln n$. 
Approximate \textsc{SetCover}

- [Slavík 1997]: Tighter analysis yields $\ln n - \ln \ln n + \Theta(1)$ as approximation factor.
- [Lund&Yannakakis (1994), Feige (1998), Moshkovitz (2015):] If $P \neq NP$ then it is impossible to devise a polynomial-time approximation algorithm for \textsc{SetCover} with approximation ratio $(1 - \alpha) \ln n$, for any constant $\alpha > 0$. 
Approximate VERTEX COVER

- VERTEX COVER can be seen as a special case of SET COVER and, thus, has an \((\ln n)\)-approximation by a simple greedy algorithm: Repeatedly delete the vertex of highest degree (and all incident edges).

- Suppose that we want to re-organize fire fighting and establish fire stations within some or all villages of a set of villages \(V := \{v_1, v_2, \ldots, v_n\}\) such that the driving distance between each village of \(V\) and its closest fire station is at most 15 km.

![Graph 1](image1)

- Hence, the greedy algorithm yields four fire stations while three stations (e.g., \(v_2, v_8, v_{10}\)) would also suffice.
Approximate VERTEX COVER: Matching

Definition 184 (Matching, Dt.: Paarung)

- A matching in a simple graph $G = (V, E)$ is a subset $E'$ of $E$ such that no two edges of $E'$ are incident upon the same vertex of $V$.
- A maximal matching is a matching that does not allow to add an additional edge.
- A maximum matching is a matching with the largest-possible number of edges.
- A perfect matching is a matching that leaves no vertex unmatched.

- Of course, a perfect matching can only exist if $G$ has an even number of nodes.
- If $G$ is weighted then we seek matchings that minimize the sum of the edge weights.
Approximate VERTEX COVER

Theorem 185

The vertices of all edges of a maximal matching of \( G \) yield a vertex cover with at most twice the number of vertices than optimum.

Proof: Let \( k_{max} \) be the number of edges of a maximum matching \( M \) of \( G \). Since every edge of \( M \) requires one vertex to cover it, any vertex cover of \( G \) contains at least \( k_{max} \) vertices.

On the other hand, we obtain a vertex cover by taking both end-points of every edge of any maximal matching of \( G \). Such a matching has \( k \) edges, with \( k \leq k_{max} \). Hence, any maximal matching of \( G \) yields a vertex cover of \( G \) with at most \( 2k_{max} \) vertices.

The sample graph has a minimum vertex cover with six vertices.
Approximate ETSP

Lemma 186 (Doubling-the-EMST heuristic)
In $O(n \log n)$ time one can achieve an approximation of ETSP for $n$ cities with approximation factor 2.

Lemma 187 (Christofides 1976)
In $O(n^3)$ time one can achieve an approximation of ETSP for $n$ cities with approximation factor $3/2$.

There exists a polynomial-time approximation scheme for solving ETSP with approximation factor $(1 + \varepsilon)$ in time $n^{O(1/\varepsilon)}$. 
Approximate ETSP

Note

- The doubling-the-EMST approach works for any complete weighted graph \( G = (V, E) \) if the weights of the edges of \( G \) satisfy the triangle inequality:

\[
c(u, v) \leq c(u, w) + c(w, v) \quad \text{for all } u, v, w \in V,
\]

where \( c(x, y) \) denotes the weight of the edge \((x, y)\).

- The TSP problem becomes much harder to approximate if we deal with settings/metrics that do not satisfy the triangle inequality! For several settings we do not have polynomial-time approximations or cannot go beyond some approximation factor, unless \( P = NP \).

- [Papadimitriou & Vempala (2000)]: For non-Euclidean TSPs with symmetric metric no polynomial-time constant-factor approximation algorithm exists which achieves \( c \leq (1 + 1/219) \), unless \( P = NP \).

- [Lampis (2012)]: For non-Euclidean TSPs with symmetric metric no polynomial-time constant-factor approximation algorithm exists which achieves \( c \leq (1 + 1/184) \), unless \( P = NP \).
Approximate ETSP: Doubling-the-EMST Heuristic

Sketch of Proof of Lem. 186:
1. Compute the Euclidean minimum spanning tree $T(S)$ of $S$.
2. Select an arbitrary node $v$ of $T(S)$ as root.
3. Compute a (pre-order-like) traversal of $T(S)$ rooted at $v$ to obtain a tour $C(S)$.
4. By-pass points already visited, thus shortening $C(S)$.
5. Apply 2-opt moves (at additional computational cost).

- Time complexity: $O(n \log n)$ for computing the EMST $T(S)$.
- Factor of approximation: $c = 2$. 
Approximate ETSP: Christofides’ Heuristic

**Sketch of Proof of Lem. 187:**

1. Compute the Euclidean minimum spanning tree $T(S)$ of $S$.
2. Get a minimum Euclidean matching $M$ on the vertices of odd degree in $T(S)$.
3. Compute an Eulerian tour $C$ on $T \cup M$.
4. By-pass points already visited, thus shortening $C$.
5. Apply 2-opt moves (at additional computational cost).

- Time complexity: $O(n^3)$ for computing $M$ [Edmonds 1965, Gabow 1972].
- Factor of approximation: $c = \frac{3}{2}$. 

\[ \]
Problems of Unknown Complexity

- There are problems which are in $\mathcal{NP}$ but which are not known to be in $\mathcal{P}$ or to be $\mathcal{NP}$-complete.

**Theorem 189 (Ladner 1975)**

If $\mathcal{P} \neq \mathcal{NP}$ then there exist problems in $\mathcal{NP}$ that are neither in $\mathcal{P}$ nor $\mathcal{NP}$-complete. These problems are called $\mathcal{NP}$-intermediate.

- Very few problems are of unknown complexity.

**Problem: GRAPH ISOMORPHISM**

- **Input:** Two (directed) graphs $G_1$ and $G_2$.
- **Decide:** Is $G_1$ isomorphic to $G_2$?

- No polynomial-time algorithm is known for the graph isomorphism problem, but the problem is also not known to be $\mathcal{NP}$-complete.

- In the end of 2015, Babai announced a deterministic algorithm that runs in time $2^{O(\log^c n)}$ time for some positive constant $c$, i.e., in quasi-polynomial time.

- [Helfgott (2017)]: Claims that $c := 3$ is fine.
Problems of Unknown Complexity

Problem: **INTEGERFACTORIZATION**

**Input:** Two numbers $n, k \in \mathbb{N}$.

**Decide:** Does $n$ have a factor less than (or greater than) some input $k$?

Problem: **DISCRETELOGARITHMPROBLEM (DLP)**

**Input:** Three numbers $a, b, k \in \mathbb{N}$.

**Decide:** Does the equation $a^x = b$ have a solution $x$ over some (finite) group that is less than or greater than some input $k$.

- [Shor (1997)]: DLP can be solved on a hypothetical quantum computer in polynomial time.

Problem: **MINIMUMCIRCUITSIZEPROBLEM (MCSP)**

**Input:** A truth table of an unknown propositional formula and a number $k \in \mathbb{N}$.

**Decide:** Does there exist a propositional formula of size $k$ that represents a given truth table?
Problems of Unknown Complexity: Minimum Convex Decomposition

**Problem: MINIMUM\textsc{ConvexDecomposition} (MCD)**

**Input:** A set $S$ of $n$ points in the plane.

**Output:** A planar straight-line graph with vertex set $S$, with each point in $S$ having positive degree, that partitions $CH(S)$ into the smallest possible number of convex faces.

- The complexity of MCD is unknown.
- [Knauer&Spillner (2006):] A 3-approximation for MCD can be computed in $O(n \log n)$ time; a $\frac{30}{11}$-approximation can be computed in $O(n^2)$ time.
- [Eder et al. (2019):] Engineering-based heuristics seem to achieve close-to-optimum solutions.
### 3SUM-Hard Problems

#### Problem: 3SUM

**Input:** A set \( S \subseteq \mathbb{Z} \) of \( n \) integers.

**Decide:** Does \( S \) contain three elements \( a, b, c \in S \) such that \( a + b + c = 0 \)?

- 3SUM was introduced by Gajentaan and Overmars (in a computational geometry paper) in 1995.
- One can solve 3SUM in \( O(n^2) \) time by means of a clever sorting-based strategy.

#### Definition 190 (3SUM-Hard)

A problem \( P \) is called 3SUM-hard if 3SUM can be reduced to \( P \) in subquadratic time.

- Several seemingly unrelated problems are known to be 3SUM-hard.
- [Grønlund&Pettie (2014):] 3SUM can be solved in \( O\left(n^2 / \left(\frac{\log n}{\log \log n}\right)^{2/3}\right) \) time!
- Still, no \( O(n^{2-\varepsilon}) \) solution is known for 3SUM, for any \( \varepsilon \in \mathbb{R}^+ \).
- [Ezra&Sharir (2017):] \( k\text{-SUM} \) has an \( O(n^2 \log^2 n) \) linear decision-tree complexity.
Linear and Integer Linear Programming

- Basics of Linear Programming
- Solving a Linear Program
- Integer Linear Programming
- Applications in CS
- Geometric and Practical Applications
Linear Program

Problem: LinearProgram (LP, Dt.: lineare Optimierung)

**Input:** Vectors \( b \in \mathbb{R}^n \) and \( c \in \mathbb{R}^d \), and a matrix \( A \in M_{n \times d} \), for \( d, n \in \mathbb{N} \).

**Output:** A solution vector \( x \in \mathbb{R}^d \) such that

\[
Ax \leq b \quad \text{and} \quad \langle c, x \rangle = \max \{ \langle c, y \rangle : y \in \mathbb{R}^d \land Ay \leq b \}.
\]

- Short-hand alternate formulations:
  - Maximize \( \langle c, x \rangle \) subject to \( Ax \leq b \).
  - Maximize \( c^T x \) subject to \( Ax \leq b \).

- We can use “minimize” instead of “maximize”, and “\( \geq \)” or “\( = \)” instead of “\( \leq \)”.

- Sample linear program:

  maximize: \( x_1 + 2x_2 \)

  subject to:

  \[
  \begin{align*}
  x_2 & \leq 4 \\
  x_2 & \geq 1 \\
  5x_1 - 4x_2 & \leq 34 \\
  x_1 + x_2 & \geq 4 \\
  3x_1 + x_2 & \leq 25
  \end{align*}
  \]

  \[
  A = \begin{pmatrix}
  0 & 1 \\
  0 & -1 \\
  5 & -4 \\
  -1 & -1 \\
  3 & 1
  \end{pmatrix}
  \]

  \[
  c^T = (1, 2) \quad x^T = (x_1, x_2)
  \]

  \[
  b^T = (4, -1, 34, -4, 25)
  \]
A company produces two types of paint: exterior and interior paint.

It makes a profit of \(€ 3000(x_1 + 2x_2)\) if it produces \(x_1\) tons of exterior paint and \(x_2\) tons of interior paint.

The set-up makes it necessary to produce at least 1 ton and at most 4 tons of interior paint per production cycle.

For production reasons, the combined output has to be at least 4 tons of paint.

One ton of exterior paint consumes 3 liters of a special liquid, while one ton of interior paint requires only 1 liter of that liquid, with 25 liters being available for one production cycle.

Marketing considerations dictate \(5x_1 - 4x_2 \leq 34\).

Sample linear program:

\[
\begin{align*}
\text{maximize:} & \quad x_1 + 2x_2 \\
\text{subject to:} & \quad x_2 \leq 4 \\
& \quad x_2 \geq 1 \\
& \quad 5x_1 - 4x_2 \leq 34 \\
& \quad x_1 + x_2 \geq 4 \\
& \quad 3x_1 + x_2 \leq 25
\end{align*}
\]

Various real-world applications, ranging from business and economics to manufacturing and engineering. E.g.:

- stock and asset management,
- transport and energy optimization,
- routing,
- scheduling and assignment planning,
- (network) flow optimization.
Sample linear program:

maximize: \( x_1 + 2x_2 \)
subject to:
\[
\begin{align*}
  x_2 & \leq 4 \\
  x_2 & \geq 1 \\
  5x_1 - 4x_2 & \leq 34 \\
  x_1 + x_2 & \geq 4 \\
  3x_1 + x_2 & \leq 25
\end{align*}
\]

Algorithm for solving LP for \( d = 2 \) manually:

1. Graph all constraints as half-planes, thus obtaining the feasible region.
2. Graph the objective function as a “movable” line.
3. Find extreme point of feasible region in direction of objective function.
**Linear Program: Basic Properties**

- Every constraint models a half-plane (for \( d = 2 \)) or a half-space (for \( d \geq 3 \)).
- The feasibility region is given by the intersection of these regions.
- Hence, the feasibility region is a convex set: It can be
  - empty,
  - unbounded,
  - bounded.
- The objective function models a line (for \( d = 2 \)) or a (hyper-)plane (for \( d \geq 3 \)).
- If the feasibility region is bounded then an optimum solution is assumed in a vertex of the feasibility region.
- Even an unbounded feasibility region may result in a unique optimum solution.

**Linear programming . . .**

. . . has nothing to do with programming in today’s meaning of the word “programming”.

**Negative variables**

Some authors and some LP codes demand non-negative variables. In such a case \( x_i \in \mathbb{R} \) can be modeled as \( x_i = x'_i - x''_i \) with \( x'_i, x''_i \in \mathbb{R}_0^+ \).
Linear Program: How to Solve It

**Simplex algorithm** by Dantzig (1947).
- Matrix manipulation based on Gaussian elimination.
- Exponential worst-case complexity — see Klee-Minty cube in $\mathbb{R}^d$ — but fast in practice.
- It remains an open question whether there is a variation of the simplex algorithm that runs in time polynomial in only $n$ and $d$.

**Ellipsoid algorithm** by Khachiyan (1979).
- First (weakly) polynomial-time algorithm for LP.
- No practical relevance.

- Runs in (weakly) polynomial time, too; quite efficient in practice.
- Several more recent IPM variants.

**Ready-to-use software:** Fierce competition between IPM and simplex methods has led to extremely fast LP solvers:
- GLPK (GNU Linear Programming Kit)
- CPLEX (IBM ILOG CPLEX Optimization Studio)
- MINOS
- GUROBI
- Mathematica, Maple, AMPL, . . .
Computation of Feasibility Region

**Theorem 191**
Randomized incremental construction allows to compute the intersection of $n$ half-planes in $\mathbb{R}^2$ in $O(n)$ expected time.

**Corollary 192**
A (bounded) LP in $\mathbb{R}^2$ with $n$ constraints can be solved in $O(n)$ expected time.

**Theorem 193 (Preparata&Muller 1979)**
The intersection of $n$ half-spaces in $\mathbb{R}^3$ can be computed (deterministically) in $O(n \log n)$ time.

- A RIC scheme applies in $\mathbb{R}^d$, too, but one needs to solve a $(d - 1)$-dimensional LP to handle the update.
- This results in an expected time that is of the form $O(d!n + \exp(d))$.
- [Clarkson 1995]: $O(d^2 n + \exp(d))$, combined with [Kalai 1992] and [Matoušek&Sharir&Welzl 1996]: $O(d^2 n + \exp(\sqrt{d \log d}))$. 
Megiddo’s Linear-Time Linear Programming

- Consider an LP for $d = 2$, i.e., in $\mathbb{R}^2$ with two variables $x$ and $y$, and $n$ constraints.
- Let $\mathcal{R}$ denote the feasibility region. W.l.o.g., we may assume that the LP amounts to seeking a feasible point with maximum $y$-coordinate: We seek $x'$ such that
  \[
  \max \{ y : (x', y) \in \mathcal{R} \} = \max \{ y : (x, y) \in \mathcal{R} \}.
  \]
- Let $\mathcal{U}$ be the set of “upper” constraints that bound $\mathcal{R}$ from above, and let $\mathcal{L}$ be the set of “lower” constraints that bound $\mathcal{R}$ from below.
- The basic idea is to compute the intersection of a line $x = \bar{x}$ with $\mathcal{R}$ in linear time, for some $\bar{x} \in \mathbb{R}$, and to decide whether $x' = \bar{x}$ or $x' < \bar{x}$ or $x' > \bar{x}$.

- Each such decision allows to transform the LP into an equivalent LP, with the same optimum but only 75% of the original constraints.
- Thus, the number of constraints is reduced from $n$ to $\frac{3}{4}n$, $(\frac{3}{4})^2 n$, $\ldots$
- An LP with a constant number of constraints is solved by brute force.
Lemma 194

One can decide in $O(n)$ time whether the line $x = \bar{x}$ intersects $\mathcal{R}$, for every $\bar{x} \in \mathbb{R}$.

Proof: Intersect $x = \bar{x}$ with all lines of $\mathcal{L}$, and let $\text{max}$ denote the $y$-coordinate of the highest intersection with all lines of $\mathcal{L}$. Similar for $\text{min}$ as the $y$-coordinate of the lowest intersection with $\mathcal{U}$.

If $\text{max} \leq \text{min}$ then $\mathcal{R}$ is not empty and $x = \bar{x}$ intersects $\mathcal{R}$. If $\text{max} > \text{min}$ then $x = \bar{x}$ does not intersect $\mathcal{R}$, or $\mathcal{R}$ is empty.
Lemma 195

1. If the line $x = \bar{x}$ intersects $\mathcal{R}$, then one can decide in $O(n)$ time whether $x' = \bar{x}$ or $x' < \bar{x}$ or $x' > \bar{x}$.

2. If the line $x = \bar{x}$ does not intersect $\mathcal{R}$, then one can decide in $O(n)$ time whether a potentially non-empty $\mathcal{R}$ has to lie to the left or to the right of $x = \bar{x}$. If this decision is not possible then $\mathcal{R}$ is guaranteed to be empty.

Sketch of Proof: This can be decided by inspecting the inclinations of the constraints of $\mathcal{L}$ and $\mathcal{R}$ that determine $\text{min}$ and $\text{max}$.
Lemma 196

Let \( \bar{x} \) be the \( x \)-coordinate of an intersection point of two constraints \( \ell_1, \ell_2 \) of \( U \). If \( \bar{x} \neq x' \) then we can drop either \( \ell_1 \) or \( \ell_2 \) from \( U \) and derive \( U' \) from \( U \), with \( U' \subset U \), such that the optimum solution of the LP remains unchanged. This process can be carried out in linear time. Same for an intersection between two constraints of \( L \).

\[ \begin{align*}
U & \quad x = \bar{x} \\
\mathcal{R} & \quad \ell_1 \\
\mathcal{L} & \quad \ell_2 \\
U & \quad x = \bar{x} \\
\mathcal{R} & \quad \ell_1 \\
\mathcal{L} & \quad \ell_2
\end{align*} \]

**Sketch of Proof:** Suppose that \( x = \bar{x} \) intersects \( \mathcal{R} \) and that \( \bar{x} \neq x' \). By Lem. 195, we can decide in \( O(n) \) time whether \( x' > \bar{x} \) or \( x' < \bar{x} \). W.l.o.g., \( x' > \bar{x} \). If \( \ell_1 \) runs above \( \ell_2 \) for all \( x > x' \) then we can drop \( \ell_1 \). Otherwise we can drop \( \ell_2 \). Same for an intersection between two constraints of \( L \). \( \square \)
Megiddo’s Linear-Time Linear Programming

Theorem 197 (Megiddo 1983,1984)

A linear program with \( n \) constraints and \( d \) variables can be solved in \( O(n) \) time when \( d \) is fixed.

Sketch of Proof for \( d = 2 \):

1. Partition the \( n_U \) constraints of \( U \) (arbitrarily) into \( \lceil n_u/2 \rceil \) pairs. Same for the \( n_L \) constraints of \( L \). (Of course, \( n_U + n_L = n \).)
2. Compute \( \lfloor n_u/2 \rfloor + \lfloor n_l/2 \rfloor \) intersection points among these pairs.
3. Choose the intersection point \((\bar{x}, y)\) whose \( x \)-coordinate \( \bar{x} \) is the median of all \( x \)-coordinates of the intersection points.
4. Use Lem. 196 to discard roughly one quarter of the constraints and recurse on the remaining roughly \( \frac{3n}{4} \) constraints.
5. Solve the LP by brute-force means for a constant number of constraints.

- Let \( T(n) \) denote the worst-case time to solve an LP with \( n \) constraints.
- We get \( T(n) = T \left( \frac{3n}{4} \right) + O(n) \), and the Master Theorem 31 implies \( T \in O(n) \).

- This deterministic algorithm is worst-case optimal but slow in practice due to large constants hidden in the \( O \)-notation.
- For \( d = 2,3 \) a linear-time solution is also due to Dyer [1984].
Integers Linear Program

Definition 198

An integer linear program (ILP) in $d$ variables $x_1, x_2, \ldots, x_d \in \mathbb{R}$ is a linear program with the additional constraint “$x_i$ is integer” for some or all $i \in \{1, 2, \ldots, d\}$.

Solving an ILP: LP relaxation [Agmon 1954]

- Drop the integer constraints and solve corresponding LP, using the same objective function and all other constraints. E.g., $x_i \in \{0, 1\}$ becomes $x_i \in \mathbb{R}$ with $0 \leq x_i \leq 1$.
- If we are lucky then the LP is “naturally integer” and will return an integer solution.
- Otherwise:
  - Apply branch & bound (aka tree search).
  - Apply branch & cut.
  - ... 

- Note: Rounding a real LP solution to an integer solution may yield a solution that is not feasible or far away from the true optimum!
- Basic practical problem: Even if the LP is solved efficiently, the subsequent transformation of the solution to make it fit the underlying ILP may be costly — it may consume exponential time!
Integer Linear Program

Standard applications of ILP

- The variables represent quantities for which fractions are meaningless, such as the number of workers or the number of busses.
- The variables represent decisions and, thus, should only take on the binary values 0 or 1.

Theorem 199

Integer linear programming is $\mathcal{NP}$-hard.

Sketch of Proof: If we could solve ILP in polynomial time then several $\mathcal{NP}$-hard problems could be solved in polynomial time. E.g., $\text{KNAP} \leq_p \text{ILP}$. 

- One can also prove that the decision version of ILP belongs to the class $\mathcal{NP}$. (But this requires some delicate arguments that a polynomial number of digits suffice.)
- In particular, the special case of 0-1 integer linear programming, in which all variables are binary, and only the restrictions must be satisfied, is one of Karp’s original 21 $\mathcal{NP}$-complete problems [Karp 1972].
Circuit Value Problem

Definition 200 (Boolean circuit, Dt.: Schaltkreis)

A Boolean circuit with \( n \) inputs and \( m \) outputs, for \( m, n \in \mathbb{N} \), is a DAG of gates of the following types:

**Input gates:** The \( n \) input gates have in-degree zero; their value is true or false.

**Logic gates:** The NOT gates have in-degree 1, while AND, OR, NAND, and NOR gates have in-degree 2. All gates follow the laws of Boolean logic. No gate has out-degree zero.

**Output gates:** Exactly \( m \) gates are output gates connected to output nodes.

Problem: CIRCUIT VALUE PROBLEM (CVP)

**Input:** A Boolean circuit with \( n \) inputs and \( m \) outputs, for \( m, n \in \mathbb{N} \).

**Output:** The output of the Boolean circuit for a given input.

- The BOOLEAN FORMULA VALUE PROBLEM is the special case of CVP when the circuit is a tree.
Circuit Value Problem as LP

Theorem 201

CVP is $P$-complete. That is, $P \leq_p \text{CVP}$ for every $P \in P$.

- $P$-complete problems are widely assumed to be inherently sequential.

Theorem 202

(The decision version of) LP is $P$-complete.

Sketch of Proof: One can reduce CVP to LP by creating a variable $x_g$ for each gate $g$, with the basic constraints $0 \leq x_g \leq 1$.

In addition we introduce the following constraints for the individual gates. E.g.:

Input gate $g$: $x_g = 1$ for $\text{true}$ and $x_g = 0$ for $\text{false}$.

NOT gate $g$ with $g \equiv \neg a$: $x_g = 1 - x_a$.

AND gate $g$ with $g \equiv a \land b$: $x_g \leq x_a$ and $x_g \leq x_b$ and $x_g \geq x_a + x_b - 1$.

OR gate $g$ with $g \equiv a \lor b$: $x_g \geq x_a$ and $x_g \geq x_b$ and $x_g \leq x_a + x_b$.

Easy to see: These constraints force all the gate variables to assume the correct values — 0 for $\text{false}$ and 1 for $\text{true}$ — and we can read off the circuit values at the variables of the output gates. (No need to maximize or minimize anything.)
Knapsack as ILP

Problem: **Knapsack (KNAP)**

**Input:** A knapsack of capacity $c \in \mathbb{N}$ and $n$ items with sizes $s_1, s_2, \ldots, s_n$ and “profits” $p_1, p_2, \ldots, p_n$.

**Output:** A subset $I$ of (the index set of) the objects that fits into the knapsack and maximizes the profit $\sum_{i \in I} p_i$.

_solution as Integer Linear Program:_ We use indicator variables $x_i \in \{0, 1\}$ for all $i \in \{1, 2, \ldots, n\}$, with $x_i = 1$ meaning that item $i$ is to be put into the knapsack.

- maximize: $\sum_{i=1}^{n} p_i x_i$
- subject to: $\sum_{i=1}^{n} s_i x_i \leq c$
- $x_i \in \mathbb{Z}$ for all $i \in \{1, 2, \ldots, d\}$
- $x_i \geq 0$ for all $i \in \{1, 2, \ldots, d\}$
- $x_i \leq 1$ for all $i \in \{1, 2, \ldots, d\}$
Problem: 3-SAT-CNF

Input: A propositional formula $A$ which is in conjunctive normal form such that every clause consists of exactly (or at most) three literals

Decide: Is $A$ satisfiable?

- Solution as Integer Linear Program: Let $z_1, z_2, \ldots, z_n$ be the Boolean variables. We use indicator variables $x_i \in \{0, 1\}$ for all $i \in \{1, 2, \ldots, n\}$, with $x_i = 1$ if and only if $z_i$ is true.

- Modeling of the clauses as constraints (for $1 \leq i < j < k \leq n$):

  
  \[
  z_i \lor z_j \lor z_k \iff x_i + x_j + x_k \geq 1
  \]

  
  \[
  \bar{z}_i \lor z_j \lor \bar{z}_k \iff (1 - x_i) + x_j + (1 - x_k) \geq 1
  \]

  
  \[
  \bar{z}_i \lor \bar{z}_j \lor \bar{z}_k \iff (1 - x_i) + (1 - x_j) + (1 - x_k) \geq 1
  \]

  
  \[
  \vdots
  \]

- Objective function: As for many decision problems, we do not have a genuine objective function — we are only interested in finding a feasible solution. We can, however, maximize 1. Or maximize $x_1 + x_2 + \ldots + x_n$. 
**INDEPENDENTSET as ILP**

**Problem: INDEPENDENTSET (IS)**

**Input:** An undirected graph $G = (V, E)$.

**Output:** A maximum independent set $I \subseteq V$. (A subset $I$ of $V$ forms an independent set of $G$ if no pair of vertices of $I$ is connected by an edge of $G$.)

- Solution as Integer Linear Program: We use indicator variables $x_v \in \{0, 1\}$ for all $v \in V$, with $x_v = 1$ meaning that node $v$ is in $I$.

  \[
  \text{maximize: } \sum_{v \in V} x_v \\
  \text{subject to: } x_v + x_w \leq 1 \text{ for all } (v, w) \in E \\
  x_v \geq 0 \text{ for all } v \in V \\
  x_v \leq 1 \text{ for all } v \in V
  \]
**Maximum Matching as ILP**

**Problem: Maximum Matching**

**Input:** An undirected graph $G = (V, E)$.

**Output:** A maximum set of edges $I \subseteq E$ such that no two edges of $I$ share the same node of $V$.

- **Solution as Integer Linear Program:** We use indicator variables $x_e \in \{0, 1\}$ for all $e \in E$, with $x_e = 1$ meaning that edge $e$ is in $I$.

  maximize:\quad \sum_{e \in E} x_e
  
  subject to:\quad \sum_{e \text{ incident to } v} x_e \leq 1 \quad \text{for all } v \in V
  
  $x_e \geq 0$ for all $e \in E$
  
  $x_e \leq 1$ for all $e \in E$

- **LP relaxation always gives an integer solution if $G$ is bipartite!**
**Problem: k-COLORING (k-COL)**

**Input:** An undirected graph $G = (V, E)$, and an integer $k \in \mathbb{N}$.

**Decide:** Does $G$ admit a coloring that uses at most $k$ colors? (An assignment of colors to all vertices of $G$ is called a (vertex) coloring if adjacent vertices are assigned different colors.)

- **Solution as Integer Linear Program:** We use indicator variables $x_{v,i} \in \{0, 1\}$ for all $v \in V$ and all colors $i \in \{1, 2, \ldots, k\}$, with $x_{v,i} = 1$ meaning that color $i$ is assigned to node $v$.

- **Constraints:**
  
  \[
  \text{maximize: } \sum_{i=1}^{k} x_{v,i} = 1 \quad \text{for all } v \in V \\
  x_{v,i} + x_{w,i} \leq 1 \quad \text{for all } (v, w) \in E \text{ and all } i \in \{1, 2, \ldots, k\} \\
  x_{v,i} \leq 1 \quad \text{for all } v \in V \text{ and all } i \in \{1, 2, \ldots, k\} \\
  x_{v,i} \geq 0 \quad \text{for all } v \in V \text{ and all } i \in \{1, 2, \ldots, k\}
  \]

- One can also apply ILP to solve COLORING.
Kernel of a Star-Shaped Polygon

**Definition 203 (Star-shaped polygon, Dt.: sternförmiges Polygon)**

A polygonal region $P$ (in the plane) is *star-shaped* if there exists a point $p \in P$ such that for every point $q \in P$ the line segment $pq$ lies entirely within $P$. The set of all points $p$ with this property is called the *kernel* (Dt.: Kern, Nucleus) of $P$.

- Hence, $P$ is star-shaped if $p$ can “see” every point on the boundary of $P$.
- Every convex polygonal region is star-shaped.

**Lemma 204**

The kernel of $P$ is not empty, and $P$ is star-shaped, if and only if the intersection of the “interior” half-planes induced by all oriented edges of the boundary polygon of $P$ is not empty.

- Formulating the problem as an LP allows to test in linear time whether $P$ is star-shaped. Furthermore, in linear time we can determine a suitable point $p$ if the kernel of $P$ is not empty.
Problem: RedBlueSeparation

**Input:** A set $R$ of red points and a set $B$ of blue points.

**Output:** A line $\ell$ such that all the red points are on one side of $\ell$ and all the blue points are on the other side, if it exists.

- All possible lines $\ell$ have the equation $a \cdot x + b \cdot y = c$, for unknown $a, b, c \in \mathbb{R}$.
- If $c \neq 0$ then $a \cdot x + b \cdot y = 1$ with $a, b \in \mathbb{R}$. Otherwise $a \cdot x + b \cdot y = 0$.
- Resulting LP, with $c := 0$ or $c := 1$:
  
  maximize: $1$

  subject to: $a \cdot x_i + b \cdot y_i \leq c$ for all $(x_i, y_i) \in R$

  $a \cdot x_i + b \cdot y_i \geq c$ for all $(x_i, y_i) \in B$

- Hence, RedBlueSeparation can be solved in time $O(|R| + |B|)$. 
Removal from Mold

Casting (Dt.: Gießen)

- A mold is a 3D shape that forms a hollow cavity.
- Liquid metal is poured into the mold (Dt.: Gussform, Kokille), and one uses gravity (or pressure) to fill the mold.
- Once the metal has solidified, the object cast is removed.
- Permanent mold casting employs reusable molds, which requires the object to be removable from the mold without destroying the mold (or the object).

- Obvious problem: Not all objects are removable from the mold.
- Can we decide efficiently whether an object can be manufactured by casting and, if so, can we find a suitable orientation of the mold and a removal direction?
- We assume that the object to be cast is formed by a polyhedron, i.e., that it is bounded by planar facets.
Removal from Mold

**Problem: MOLD REMOVAL**

**Input:** A polyhedral object \( P \), with designated (horizontal) top facet, and its corresponding mold.

**Output:** A direction vector \( d \in \mathbb{R}^3 \), if it exists, such that \( P \) can be translated to infinity in direction \( d \) without intersecting the mold.

- A facet of \( P \) is called *ordinary facet* if it is not the top facet of \( P \).

**Lemma 205**

The polyhedron \( P \) can be removed from its mold by a translation in direction \( d \) if and only if \( d \) forms an angle of at least 90° with the outward normals of all ordinary facets of \( P \).

**Lemma 206**

The direction vector \( d := (d_x, d_y, d_z) \) forms an angle of at least 90° with an outward normal vector \( v := (v_x, v_y, v_z) \) if and only if

\[
\langle d, v \rangle := d_x \cdot v_x + d_y \cdot v_y + d_z \cdot v_z \leq 0.
\]
Removal from Mold

Lemma 207

**MOLD REMOVAL** can be solved in $O(n)$ time for a polyhedron $P$ with $n$ ordinary facets.

**Proof:** Since the direction vector $d$ must have a non-zero $z$-coordinate, we can write $d$ as $(d_x, d_y, 1)$.

Let $v_i := (v_{ix}^i, v_{iy}^i, v_{iz}^i)$ be the outward normal vector of the $i$-th ordinary facet of $P$, for $i \in \{1, 2, \ldots, n\}$. Then we get the following simple LP:

\[
\begin{align*}
\text{minimize:} & \quad d_x + d_y \quad \text{(or simply 1)} \\
\text{subject to:} & \quad v_{ix}^1 \cdot d_x + v_{iy}^1 \cdot d_y + v_{iz}^1 \leq 0 \\
& \quad v_{ix}^2 \cdot d_x + v_{iy}^2 \cdot d_y + v_{iz}^2 \leq 0 \\
& \quad \vdots \\
& \quad v_{ix}^n \cdot d_x + v_{iy}^n \cdot d_y + v_{iz}^n \leq 0
\end{align*}
\]

A removal direction $d$ for separating the polyhedron from its mold exists if and only if this LP is feasible, which can be solved in $O(n)$ time.
Test for Roundness

- A circular annulus (in $\mathbb{R}^2$) is the region between two concentric circles. It is determined by a center $c := (c_x, c_y)$ and two radii, $r$ and $R$, with $r \leq R$.

**Problem: MINIMUM AREA ANNULUS**

**Input:** A set $S$ of points in $\mathbb{R}^2$.

**Output:** A minimum-area circular annulus that contains all points of $S$.

- MINIMUM AREA ANNULUS can be used to check the “roundness” of a 2D shape by determining the cocircularity of points probed on the shape.

Hence, we seek to solve the following optimization problem:

minimize: $\pi(R^2 - r^2)$

subject to: $r \leq \|c - p\| \leq R$ for all $p \in S$

- This does not look like a linear problem, does it?
Test for Roundness as LP

- We introduce two new variables $u, v \in \mathbb{R}$ and a new constraint:
  \[
  u := r^2 - \|c\|^2 \quad \text{and} \quad v := R^2 - \|c\|^2 \quad \text{and} \quad v \geq u
  \]
- This transforms the objective function into $\pi(v - u)$, or simply into $v - u$.
- We can re-write the constraints as $r^2 \leq \|c - p\|^2 \leq R^2$.
- Since $\|c - p\|^2 = \|c\|^2 - 2\langle c, p \rangle + \|p\|^2$ for all $p \in S$, we have
  \[
  r^2 \leq \|c - p\|^2 \iff r^2 \leq \|c\|^2 - 2\langle c, p \rangle + \|p\|^2 \iff u + 2\langle c, p \rangle \leq \|p\|^2.
  \]
- Similarly,
  \[
  \|c - p\|^2 \leq R^2 \iff v + 2\langle c, p \rangle \geq \|p\|^2.
  \]
- This transforms the objective function and the constraints into a linear program in $u, v, c_x, c_y$, with the new constraint $v \geq u$.
- We can use $r^2 = u + \|c\|^2$ and $R^2 = v + \|c\|^2$ to reconstruct $r$ and $R$.
- It cannot happen that $r^2$ turns out to be negative in an optimum solution: Since we only demand $r^2 \leq \|c - p\|^2$ for all $p \in S$, we could definitely push $r^2$ to at least 0 by increasing $u$, while also making the objective $v - u$ smaller!
The End!

I hope that you enjoyed this course, and I wish you all the best for your future studies.