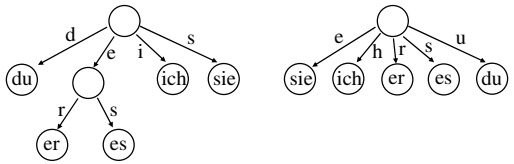
8.6 Digitale Suchbäume

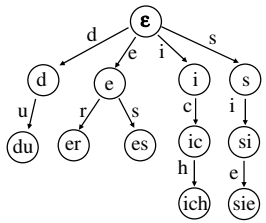
Sind Schlüssel Wörter über einem Alphabet (und das sind sie meistens), so kann man sie zeichenweise (digit per digit, also "digital") lesen und hiermit gleichzeitig einen Baum durchlaufen, entweder vollständig oder bis die restliche Zeichenfolge eindeutig auf den Schlüssel schließen lässt.

Beispiel: Die Schlüssel lauten "ich", "du", "er", "sie", "es".



Durchlaufen der Schlüssel von vorne (links) und hinten (rechts).

Baum mit vollständigen Pfaden (von vorne nach hinten durchlaufener Schlüssel):
Die Schlüssel lauten wieder "ich", "du", "er", "sie", "es".

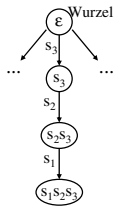


Einen solchen geordneten digitalen Suchbaum nennt man einen Trie.

Meist fasst man die Schlüssel s als Zahl zur Basis m auf, also $s \in \Sigma^* = \{0, 1, ..., m\text{-}1\}^*$; m ist die Zahl der Alphabetzeichen. Betrachte $s = s_1 s_2 ... s_k$, dann erhält man den Knoten, der zu s gehört, indem man die Zeichen von hinten nach vorne mittels Modulo-Bildung berechnet:

```
z := "Wurzel des Baums"; zahl := s;
for i := 1 to k do                    -- oder: while zahl > 0 do ...
        x := zahl mod m;  zahl := zahl div m;
        z := "x-tes Kind von z"  od;
"Untersuche den Inhalt von z"
```


Der Pfad von der Wurzel zu einem Knoten mit dem Schlüssel $s_1 s_2 ... s_k$ ist mit genau diesen Zeichen s_i markiert. Dabei kann man Abkürzungen vornehmen, sofern die weitere Zeichenfolge eindeutig (= eine Kantenfolge ohne Abzweigungen) ist. *Wir lesen hier die Schlüssel von hinten, da sie auch in dieser Reihenfolge berechnet werden!*

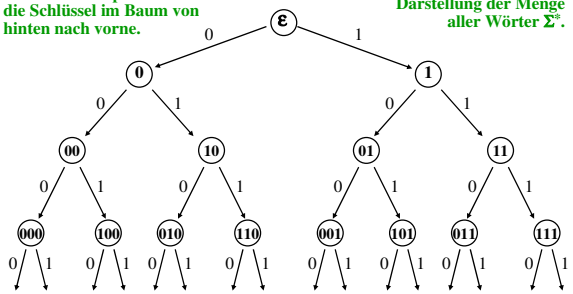


Viele Knoten dienen nur als Verzweigungsknoten zu den Knoten der Schlüssel. Prinzipiell muss man in jedem Knoten mit so vielen Söhnen rechnen, wie es Zeichen im Alphabet gibt. Z.B.: Buchstaben und Ziffern für Bezeichner oder alle Nicht-Steuerzeichen des ASCII-Alphabets (62 bzw. 96 Elemente).

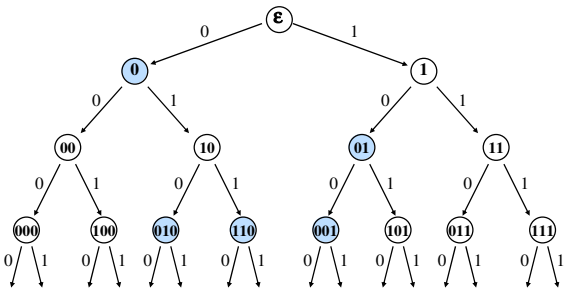
Am Beispiel des Alphabets $\Sigma = \{0, 1\}$ lässt sich dies gut demonstrieren.

Beachte: Wir speichern hier die Schlüssel im Baum von hinten nach vorne.

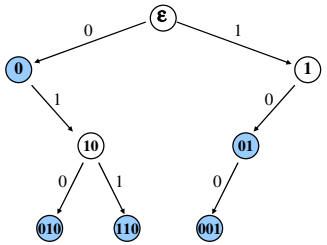
Darstellung der Menge aller Wörter Σ^*.



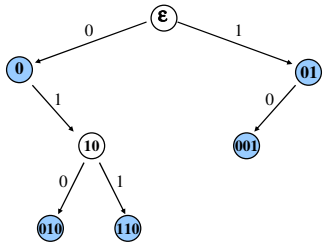
Jeder Knoten entspricht der Beschriftung auf dem Pfad von der Wurzel zu ihm. 0 = linkes Kind, 1 = rechtes Kind.



Die Menge {0, 01, 010, 110, 001} im unendlichen Baum.



Die Menge {0, 01, 010, 110, 001} als digitaler Baum.



Etwas verkürzter Baum für die Menge {0, 01, 010, 110, 001}. Beim Suchen und Einfügen muss man dann ab dem aktuell im Knoten stehenden Schlüssel weiter vergleichen.

Digitaler Suchbaum über {0,1}:

Ansatz 1: Gegeben sind Schlüssel als 0-1-Folgen. Für jeden solchen Schlüssel durchlaufe man den (binären) 0-1-Baum entsprechend der Binärfolge des Schlüssels und platziere ihn genau in dem zum Schlüssel gehörenden Knoten.

FIND, INSERT und DELETE verhalten sich nun wie bei binären Suchbäumen.

Vorteile dieser Struktur vor allem dann, wenn man auf Maschinenebene programmiert oder die Binärdarstellung von Schlüsseln vorliegt.

Nachteil: Dies kostet oft viel Speicherplatz.

Digitaler Suchbaum über {0,1}:

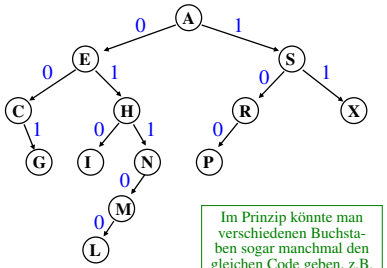
Ansatz 2: Verkürzung der Pfade um lineare Ketten. Gegeben sind Schlüssel als 0-1-Folgen. Für jeden solchen (stets vorwärts- oder stets rückwärts gelesenen) Schlüssel durchlaufe man den 0-1-Baum und platziere den Schlüssel auf den ersten freien Knoten.

FIND, INSERT und DELETE wie bei binären Suchbäumen, *jedoch muss in jedem Knoten auf Gleichheit während des Durchlaufs geprüft werden.*

Digitale Suchbäume über beliebigen Alphabeten ergeben sich als nahe liegende Verallgemeinerung.

Übliches Vorgehen: Codiert man Zeichen eines beliebigen Alphabets binär, so kann man entsprechend der Reihenfolge der Schlüssel mit ihnen einen Binärbaum aufbauen und jeden Schlüssel in das jeweils erreichte Blatt legen. Beispiel (aus der Plödereder-Vorlesung 2001) *diesmal von vorne nach hinten:*

A: 00001
S: 10011
E: 00101
R: 10010
C: 00011
H: 01000
I: 01001
N: 01110
G: 00111
X: 11000
M: 01101
P: 10000
L: 01100



Im Prinzip könnte man verschiedenen Buchstaben sogar manchmal den gleichen Code geben, z.B. P↔10010 (↔R).

In der Praxis verwendet man ein Endezeichen für Schlüssel, z.B. '#'.

Die Schlüssel 1, 10 und 101 werden also als
1#, 10# und 101# dargestellt,
wenn man einen digitalen Baum aufbaut.

Die Implementierung dieser Ideen sollte nun klar sein (?). Hinweise zur möglichen Datenstruktur für digitale Bäume über {0,1} und zum Suchen und Aufbauen finden Sie auf den nächsten Folien. Das Löschen in einem digitalen Baum ist je nach Ansatz unterschiedlich aufwändig.


```
Maxlaenge: constant Positive := ... ;
type Stelle is (0,1,#);
type Schluesseltyp is array (1..Maxlaenge) of Stelle;
type Knotentyp;
type Baumtyp is access Knotentyp;
type Knotentyp is record
        Schluessel: Schluesseltyp := (others => #);
        L, R: Baumtyp;
        end record;
```

Für eine natürliche Zahl (= Schlüssel) s sei
bit(s,k) = if $s < 2^k$ then k-tes Bit von hinten in der Binärdarstellung von s else # fi.
bit(s,1) ist also die letzte Binärstelle von s, bit(s,2) die vorletzte usw. Vorne wird mit '#' aufgefüllt (nicht mit 0).

Suche oder Einfügen für Schlüssel s (nach Ansatz 2)

```
h1, h2: Baumtyp; i: Natural := Maxlaenge; Code: Schluesseltyp;
begin  h1 := "Verweis auf den digitalen Suchbaum"; h2 := null;
    "wandle s mittels bit(s,k) für k = 1, 2, ..., Maxlaenge in
        Schluesseltyp um und weise dies Code zu";
    while h1 /= null and then h1.Schluessel /= Code loop
        h2 := h1;
        if Code(i) = 0 then h1:=h1.L;
        elsif Code(i) = 1 then h1:=h1.R;
        else h1 := null; end if;
        i:= i-1;
    end loop;
    if h1 = null then "s nicht im Baum; h2 verweist auf den Knoten,
        an den ein Knoten mit Schlüssel s angehängt werden kann"
    else "h1 verweist auf Knoten mit dem Schlüssel s" end if;
end;
```

Man hört hierbei auch auf, wenn '#' erreicht wird.

8.7 Datenstrukturen mit Historie

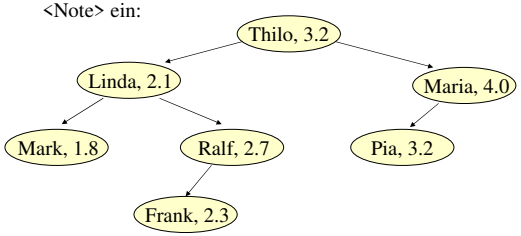
Dieser Abschnitt entstand nach einem Vortrag von Prof. Thomas Ottmann (Universität Freiburg) am 25.6.2004.

Entwicklungen und Vorgänge unterliegen der Zeit. So gibt es auch beim Aufbau einer konkreten Datenstruktur stets eine Historie, also einen zeitlichen Ablauf, in dem diese Struktur entsteht.

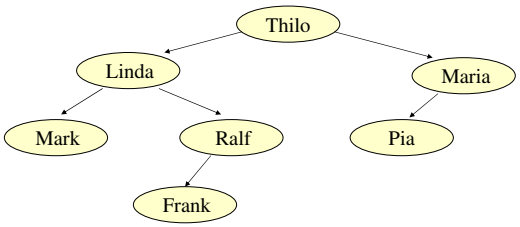
Wenn wir uns die zeitliche Reihenfolge merken möchten, so müssen wir die Datenstruktur und die verwendeten Operationen kennen. In diesem Kapitel 8 würden wir geordnete Bäume und die Operationen FIND, INSERT und DELETE wählen.

Wir betrachten ein Beispiel:

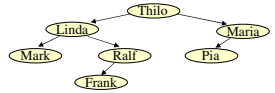
8.7.1 Beispiel "Teilnehmer und Ergebnisse einer Klausur":
Die Datenstruktur sei ein binärer Suchbaum. Die verwendeten Daten "(<Name>,<Note>)" bilden die Menge
{(Tilo, 3.2), (Linda,2.1), (Ralf, 2.7), (Maria, 4.0), (Mark, 1.8), (Frank,2.3), (Pia, 3.2)}.
Diese 7 Elemente tragen wir in dieser Reihenfolge mittels INSERT in einen binären Suchbaum bzgl. des Schlüssels <Note> ein:



Wir möchten nun wissen, wer an der Klausur teilgenommen hat. Hierzu lassen wir die Noten weg und übermitteln den verbliebenen Baum.



Können wir diesem fertigen Produkt, also dem binären Baum, mehr als nur die Teilnehmer ansehen? **Ja!**



Erstens erkennen wir die Rangordnung der Klausurergebnisse mit einem Inorder-Durchlauf:
Mark, Linda, Frank, Ralf, Thilo, Pia, Maria.

Zweitens liefert uns der Baum eine Halbordnung der Eintragungsreihenfolge: Jeder Pfad von der Wurzel zu einem Blatt gibt die relative Reihenfolge beim Einfügen an.
So müssen Linda vor Ralf und Frank, Maria vor Pia, Linda vor Mark usw. eingefügt worden sein.

Folgerung: Die Datenstruktur speichert also nicht nur irgendwelche Fakten, sondern sie merkt sich zugleich Ausschnitte aus ihrer Historie und gewisse zusätzliche Inhalte.

Im hier betrachteten Fall sind die zusätzlichen Informationen nichts anderes als Relationen auf der Menge der Daten.

Idee und intuitive Formulierung: Es sei M die Menge der Daten, die in einer Datenstruktur D gespeichert sind. Eine Relation $R \subseteq M^k = M \times M \times ... \times M$ heißt *mit D verträglich,* wenn jede Beziehung $(m_1, m_2, ..., m_k) \in R$ aus der Datenstruktur D gefolgert werden kann.

Diese Formulierung ist nicht so genau, dass wir sie programmieren könnten. Es fehlt die Präzisierung, was es bedeutet, dass eine Beziehung aus einer Datenstruktur gefolgert werden kann. Hierzu müssten wir eine formale Logik über den Datenstrukturen definieren. Siehe hierzu Vorlesungen aus dem Gebiet "sichere und zuverlässige Systeme". Wir verfolgen daher eine andere Idee.

Eine Datenstruktur d_i entsteht aus der Menge der Daten M, indem zulässige Operationen auf Elemente aus M und die bis dahin gewonnene Datenstruktur d_{i-1} angewendet werden. Anfangs ist die Datenstruktur leer: d_0 = 'empty'.

Definition 8.7.2
- Es sei M eine Menge. Es sei D eine Menge von Datenstrukturen, deren Inhalte aus M seien. Es sei 'empty' eine spezielle Datenstruktur aus D (die "leere Struktur").
- Es sei Ω eine endliche Menge von Operationen.
- Der Einfachheit halber nehmen wir hier an, dass für jedes $\omega \in \Omega$ gilt: $\omega: M \times D \to D$.

Eine Folge von Operationen und Elementen $h = (\omega_1, m_1), (\omega_2, m_2), ..., (\omega_r, m_r)$ heißt eine **Historie** der Länge r von $d = \omega_r(m_r, ...(\omega_2(m_2, \omega_1(m_1, \text{empty})))...) \in D$.

Wir vollziehen also den Aufbau der Datenstruktur schrittweise nach. Wenn wir nacheinander die Operationen $\omega_1, \omega_2, ..., \omega_r$ mit den Elementen $m_1, m_2, ..., m_r$ durchführen, so erhalten wir nacheinander die Datenstrukturen $d_0, d_1, d_2, ..., d_r$:

d_0 = 'empty', $d_1 = \omega_1(m_1, \text{empty})$, $d_2 = \omega_2(m_2, d_1)$, $d_3 = \omega_3(m_3, d_2)$, ... $d_i = \omega_i(m_i, d_{i-1})$ für alle $i > 0$, ... ,
und am Ende:
$d_r = \omega_r(m_r, d_{r-1}) = \omega_r(m_r, ...(\omega_2(m_2, \omega_1(m_1, \text{empty})))...)$.

Frage 1: Wenn man d_r kennt, was lässt sich dann über dessen Historie $(\omega_1,m_1), (\omega_2,m_2), ..., (\omega_r,m_r)$ aussagen?

Frage 2: Wieviele verschiedene Historien (der Länge r oder ≤r) kann eine Datenstruktur $d \in D$ haben?

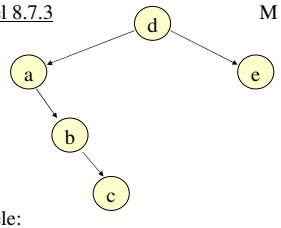
Frage 3: Was muss man tun, um aus d stets auf die Historie schließen zu können?

Dies muss man an Beispielen erläutern. Wir wählen als Menge M die ganzen Zahlen **Z** und als Menge D die Menge der binären (Such-) Bäume mit ganzen Zahlen als Inhalten. 'empty' sei der leere Baum.

Als endliche Menge der Operationen Ω wählen wir {FIND, INSERT, DELETE}. Für jedes $\omega \in \Omega$ ist $\omega: M \times D \to D$ klar:
- im Falle von FIND nehmen wir die Identität,
- im Falle von INSERT wird das Element als Blatt eingefügt,
- im Falle von DELETE wird über den Inorder-Nachfolger gelöscht, vgl. 8.2.10.

Wir betrachten nun einen binären Baum und fragen danach, welche Historie zu ihm gehört. Dies ist im Allgemeinen nicht eindeutig. Daher interessiert uns die Anzahl aller Historien, die ein solcher Baum haben kann.

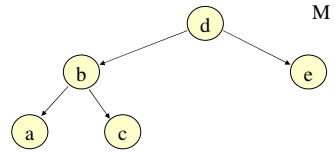
Beispiel 8.7.3



M = {a, b, c, d, e}

Dieser Baum besitzt genau 4 Historien, wenn man nur die Operation IN (= INSERT) verwenden darf.

Beispiele:
Historie 1: (IN,d), (IN,a), (IN,b), (IN,c), (IN,e)
Historie 2: (IN,d), (IN,a), (IN,e), (IN,b), (IN,c)
Hinweis: Die Anzahl der Historien, die *nur* die INSERT-Operation verwenden, ist gleich der Zahl der toplogischen Sortierungen des Baums (hierzu vgl. Anhang 8.8.2).
Historie 3: (IN,c), (IN,d), (IN,e), (DEL,c), (IN,a), (IN,s), (IN,c)



M = {a, b, c, d, e}

Historie 1: (IN,d), (IN,b), (IN,a), (IN,c), (IN,e)
Historie 2: (IN,d), (IN,b), (IN,e), (IN,c), (IN,a) usw.
Dieser Baum besitzt genau 8 Historien, wenn man nur die Operation IN verwenden darf. Seine Entstehung ist also "unbestimmter" als die des Baums auf der vorherigen Folie. Unbestimmheit bezeichnet man oft als Entropie.

Definition 8.7.4:

Es seien M, D, Ω und $\omega: M \times D \to D$ (für $\omega \in \Omega$) wie in Def. 8.7.2.
Für eine natürliche Zahl r und eine Struktur $d \in D$ sei die r-Entropie von d die Anzahl der Historien der Länge r, die zu d gehören.
Sofern es zu d nur endlich viele Historien gibt, bezeichnet man die Anzahl aller Historien als die Entropie von d.

Hinweis: Die Strukturen mit hoher Entropie sind also zugleich die "vergesslichen Strukturen", in denen die eigene Entstehung nur teilweise notiert werden kann.

Fallstudie 8.7.5:

Es seien $M = \mathbf{Z}$ die Menge der ganzen Zahlen, D = die Menge aller binären (Such-) Bäume (mit Inhalten aus **Z**, ⇒ 8.2.10), $\Omega = \{IN\}$ eine einelementige Menge und IN: $\mathbf{Z} \times D \to D$ die übliche Einfügeoperation INSERT, die durch die Prozedur procedure Einfügen(Anker: in out Ref_BinBaum; s: Integer) in 8.2.10.b exakt definiert ist (der neue Schlüssel $s \in \mathbf{Z}$ wird hierbei in ein neues Blatt eingefügt).

Gegeben sei eine natürliche Zahl n und die Menge der ersten n natürlichen Zahlen $\underline{\mathbf{n}} := \{0, 1, 2, ..., n-1\} \subset \mathbf{Z}$. Betrachte die Menge D_n alle binären Bäume mit genau den n verschiedenen Inhalten 0, 1, 2, ..., n-1. Dann bildet die Menge $S_n = \{i_1 i_2 ... i_n \mid i_j \in \underline{\mathbf{n}}, i_j \neq i_k \text{ für } j \neq k\}$ (= die Menge aller Anordnungen der n ersten natürlichen Zahlen) genau die Menge der Historien aller dieser binären Bäume.

Jeder Anordnung $i_1 i_2 ... i_n$ ist genau ein binärer (Such-) Baum zugeordnet, der durch Einfügen hieraus entsteht und zu dessen Historie sie gehört. Es gibt also eine (surjektive) Abbildung $\varphi: S_n \to D_n$.

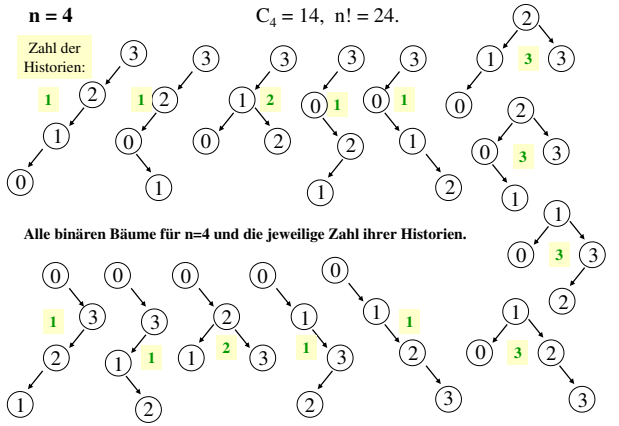
Formal genauer: Die Historie $h = (IN,i_1), (IN,i_2), ..., (IN,i_n)$, die wir mit der Folge $i_1 i_2 ... i_n$ identifizieren (mit $i_j \neq i_k$ für $j \neq k$), liefert genau einen binären Baum $\varphi(h)$.

Für die inverse Abbildung $\varphi^{-1}: D_n \to 2^{S_n}$ gilt dann:
$|\varphi^{-1}(d)|$ ist die Entropie des binären Baums $d \in D_n$.
$(\varphi^{-1}(d) = \{ i_1 i_2 ... i_n \mid \varphi(i_1 i_2 ... i_n) = d \} \subset S_n.)$

Faustregel: Je gleichverzweigter der Baum d ist, umso größer ist seine Entropie.

Dies beleuchten wir zunächst am Beispiel n=4.

n = 4 $C_4 = 14$, $n! = 24$.


Zahl der Historien:
1
1
2
3
3
3

Anhang zu Kapitel 8:

8.8 Weitere Definitionen zu Graphen

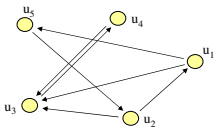
8.9 Sonstiges

8.8 Weitere Definitionen zu Graphen

Graphen wurden in Abschnitt 3.8 auf 36 Folien ausführlich erläutert. Gehen Sie bitte jene Folien nochmals genau durch. Die Begriffe gerichteter und ungerichteter Graph, adjazent bzw. benachbart, inzident, (induzierter) Teilgraph, Grad, Weg, Kreis, Länge von Wegen, erreichbar, azyklisch, starke und schwache Zusammenhangskomponente, Adjazenzmatrix, Adjazenzliste (mit zugehöriger Datenstruktur), transitive Hülle und Graphdurchlauf sollten Ihnen gut vertraut sein.

Auf den folgenden Seiten ergänzen wir jene Definitionen um Begriffe, die zum Teil in diesem Kapitel aufgetreten sind. Wir werden sie in den Übungen vertiefen. Hiermit werden zugleich die Graphalgorithmen in Kapitel 11 vorbereitet.

8.8.1 Erinnerung zur Darstellung von Graphen:
Graphen kann man durch ihre *Adjazenzmatrix* A, durch *Adjazenzlisten* oder durch *Inzidenzlisten* darstellen (siehe 3.8.5 g).



Diesen Graphen stellen wir als Adjazenzmatrix und als Adjazenzliste dar.

$$\begin{pmatrix} 0&0&1&0&1\\ 1&0&1&0&0\\ 0&0&0&1&0\\ 0&0&1&0&0\\ 0&1&0&0&0 \end{pmatrix}$$

Adjazenzmatrix A

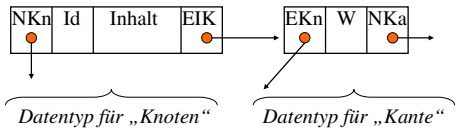
$$\begin{pmatrix} 1&0&1&0&1\\ 1&1&1&0&0\\ 0&0&1&1&0\\ 0&0&1&1&0\\ 0&1&0&0&1 \end{pmatrix}$$

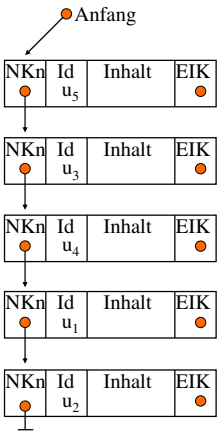
Erweiterte Adjazenzmatrix A'

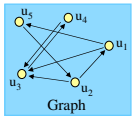
Darstellung als Adjazenzliste (siehe 3.8.6)

Jeder *Knoten* erhält einen Identifikator Id (z.B. einen Bezeichner oder eine natürliche Zahl) und einen Inhalt. Die Knoten werden in einer Liste zusammengefasst und besitzen neben Id und Inhalt weitere Komponenten:
- Verweis auf den **N**ächsten **K**noten in der Liste "NKn",
- Verweis auf die **E**rste **I**nzidente **K**ante "EIK".

Jede von einem Knoten ausgehende *Kante* muss enthalten: den "Endknoten der Kante" (EKn), ihren Wert W ("weight") und einen Verweis auf die nächste Kante "NKa".



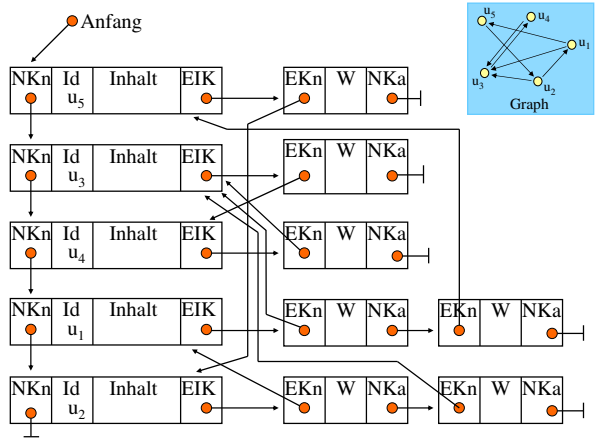




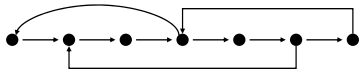
Zunächst werden die Knoten in beliebiger Reihenfolge in einer Liste gespeichert (siehe links).

Danach (siehe nächste Folie) werden die Kanten über die EIK-Listen eingetragen.

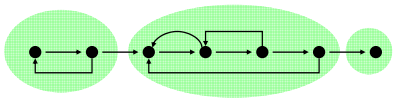
Wir fügen hier keine Werte für die Komponenten „Inhalt“ und „W“ ein.



Erinnerung: Zusammenhang im gerichteten Fall



Dieser Graph ist stark zusammenhängend.



Dieser Graph besitzt drei starke Zusammenhangskomponenten.

Vgl. hierzu 3.8.14.

Definition 8.8.2: Topologische Sortierung von DAGs

Zu einem Graphen $G=(V, E)$ heißt $G^*=(V, E^*)$ die transitive Hülle, wenn im ungerichteten Fall gilt
$E^* = \{\{u,v\} \mid u \neq v$ und es gibt einen Weg von u nach v in $G\}$
bzw. im gerichteten Fall gilt
$E^* = \{(u,v) \mid u \neq v$ und es gibt einen Weg von u nach v in $G\}$.
Stets gilt natürlich $E \subseteq E^*$.

Es sei $G=(V,E)$ gerichtet. Eine Abbildung ord: $V \to \mathbb{N}$ mit
$\forall\, u, v \in V$ mit $u \neq v$ gilt: $(u,v) \in E^* \Rightarrow \text{ord}(u) < \text{ord}(v)$
heißt topologische Sortierung von G.
Man ordnet also die Knoten so an, dass jeder von u aus erreichbare Knoten eine höhere Nummer als u bekommt.

Genau jeder azyklische gerichtete Graph („DAG“) besitzt eine topologische Sortierung; sie ist nicht eindeutig (selbst am Beispiel klar machen!).

Definition 8.8.3: Markierte oder gewichtete Graphen

In Anwendungen sind Graphen meist "markiert" oder "gewichtet", d.h., ihre Knoten und/oder ihre Kanten sind mit Werten ("Markierung" oder "Gewicht") aus einer Wertemenge W bzw. W' versehen: $\mu: V \to W$ und $\delta: E \to W'$.

Man schreibt $G=(V, E, \mu)$ bzw. $G=(V, E, \delta)$ bzw. $G=(V, E, \mu, \delta)$. Meist sind die Markierungen ganze oder reelle Zahlen. Oft hat man mehr als nur zwei solche Abbildungen.

Kantenmarkierungen $\delta: E \to \mathbb{R}^{\geq 0}$ mit
$\mathbb{R}^{\geq 0}$ = Menge der nichtnegativen reellen Zahlen
bezeichnet man auch als Entfernungen.

[Die Summe aller Entfernungen nennt man das Gewicht des Graphens, siehe unten bei "minimaler Spannbaum".]

Definition 8.8.4: Länge von Wegen bzgl. δ, Abstand

Sei $G=(V,E,\delta)$ ein Graph. Die reellwertige Abbildung $\delta: E \to \mathbb{R}$ wird auf die Menge der Wege fortgesetzt durch $\delta((u_0,u_1,...,u_k)) := \delta((u_0,u_1)) + \delta((u_1,u_2)) + ... + \delta((u_{k-1},u_k))$. Dieser Wert heißt die Länge des Weges $(u_0,u_1,...,u_k)$.

Sei $\delta: E \to \mathbb{R}^{\geq 0}$ (= Menge der nichtnegativen reellen Zahlen), dann bezeichnet man für den Graphen $G=(V,E,\delta)$ die (kürzeste) Entfernung oder den Abstand oder die Distanz eines Knotens u zum Knoten v den Wert $\delta: V\times V \to \mathbb{R}^{\geq 0} \cup \{\infty\}$ mit

$\delta(u,v) = 0$, für $u=v$,
$\delta(u,v) = \text{Min}\,\{\delta((u_0,u_1,...,u_k)) \mid u=u_0, v=u_k\}$, sofern es mindestens einen Weg von u nach v gibt,
$\delta(u,v) = \infty$, falls es keinen Weg von u nach v gibt.

Statt $\delta(u,v)$ schreibt man oft dist(u,v) oder d(u,v).
Die maximale Distanz in einem Graphen bezeichnet man auch als den Durchmesser des Graphen.

8.8.5: Kürzeste Wege

Die Aufgabe, den Abstand dist(u,v) vom Knoten u zum Knoten v und einen Weg von u nach v mit der Länge dist(u,v) zu ermitteln, bezeichnet man als das Kürzeste-Wege-Problem.

Hierbei unterscheidet man die Probleme:

SSSP = single source shortest paths
= alle kürzesten Wege, die von einem gegebenen Knoten zu jedem anderen Knoten führen,

SPSP = single pair shortest path
= kürzester Weg zwischen zwei gegebenen Knoten u und v

APSP = all pair shortest paths
= alle kürzesten Wege zwischen allen Paaren (u,v) von Knoten

Definition 8.8.6: (minimaler) Spannbaum, Gewicht

Sei $G=(V,E)$ ein zusammenhängender gerichteter oder ungerichteter Graph. Ein (gerichteter bzw. ungerichteter) Teilgraph $B=(V,E_B)$, der ein Baum ist, heißt Spannbaum oder aufspannender oder spannender Baum von G (beachte: in B kommen alle Knoten des Graphens vor).

Es sei $G=(V,E,\delta)$ ein Kanten-markierter Graph mit $\delta: E \to \mathbb{R}$. Die Summe aller seiner Entfernungen $\delta(G)$

$$\delta(G) := \sum_{e \in E} \delta(e)$$

heißt das Gewicht des Graphen G.

Ein Spannbaum $B=(V,E_B,\delta)$ des Graphen $G=(V,E,\delta)$ heißt minimaler Spannbaum (engl.: minimal spanning tree) von G, wenn $\delta(B) \leq \delta(B')$ für alle Spannbäume B' von G gilt.

Definition 8.8.7: Hamiltonsche und Eulersche Wege

Ein Weg $(u_0,u_1,...,u_{n-1})$ in einem Graphen G mit n Knoten heißt Hamiltonscher Weg, wenn er jeden Knoten genau einmal enthält. Ein Weg $(u_0,u_1,...,u_{n-1},u_0)$ heißt Hamiltonscher Kreis, wenn $(u_0,u_1,...,u_{n-1})$ ein Hamiltonscher Weg ist.

Ein Weg $(u_0,u_1,...,u_k)$ heißt Eulerscher Weg, wenn jede Kante des Graphen genau einmal in ihm vorkommt. Er heißt Eulerscher Kreis, wenn zusätzlich $u_k=u_0$ ist.

Die Bestimmung Hamiltonscher Wege ist schwierig, diejenige Eulerscher Wege dagegen leicht, siehe unten.

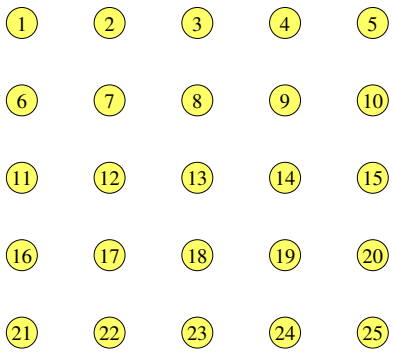
8.8.8 Einige Fragen und ihre Lösungen.

Wir wollen folgende Aussagen erläutern.

1. Die Zahl der verschiedenen (doppelpunktfreien) Wege zwischen zwei festen Knoten u und v in einem Graphen mit n Knoten und höchstens 2n Kanten kann bereits exponentiell wachsen. Wir geben ein Beispiel mit mehr als $4^{\sqrt{n}}$ solchen Wegen an („Gitter“).

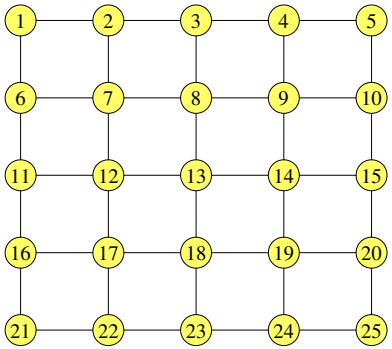
2. Hamiltonsche Wege/Kreise sind nicht leicht zu finden. (Manche von Ihnen werden später beweisen, dass dieses Problem „NP-hart“ ist und daher nach heutiger Meinung nur mit exponentiellem Aufwand gelöst werden kann.)

3. Eulersche Wege/Kreise sind dagegen leicht zu finden. (Wir erläutern dies nur an einem Beispiel. Den Beweis des Satzes sollten Sie selbst versuchen oder in einem Lehrbuch nachlesen.)



Wir betrachten im Folgenden diese Knotenmenge {1, 2, 3, ... , 25}.

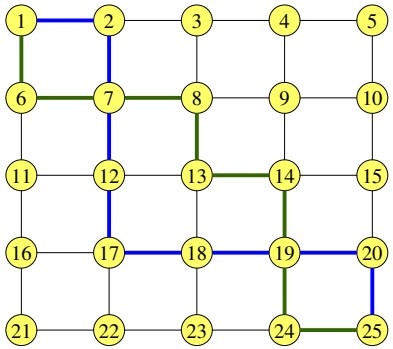
zu 1. Berechne die Anzahl von Wegen zwischen zwei Knoten



Als Beispiel wählen wir ein ungerichtetes „Gitter“.

Knoten: 25.
Kanten: 40.

Wie viele verschiedene Wege gibt es von Knoten 1 nach Knoten 25?



Knoten: 25.
Kanten: 40.

Es sind *mindestens*

$$\binom{10}{5} = (10\cdot9\cdot8\cdot7\cdot6) / (1\cdot2\cdot3\cdot4\cdot5) = 252 \text{ Wege.}$$

Allgemein: Wenn ein k×k-Gitter mit $k^2 = n$ vielen Knoten gegeben ist, so gibt es mindestens $\binom{2k}{k}$ doppelpunktfreie Wege zwischen den beiden Knoten 1 und k^2.

Warum? Man kann genau k-mal eine Kante nach rechts („r“) und genau k-mal eine Kante nach unten („u“) gehen. Die Reihenfolge ist beliebig. Dies gibt eine Folge der Länge 2k bestehend aus je k Buchstaben „r“ und „u“. Man kann aus den 2k Positionen beliebig k Stellen für den Buchstaben „r“ auswählen (auf die restlichen Stellen kommt dann der Buchstabe „u“). Man erhält genau „2k über k“ Möglichkeiten. Dies ist aber nur ein Bruchteil der tatsächlichen Möglichkeiten, wie man sich leicht überlegt.

Man beachte: Die Zahl der Wege wächst exponentiell mit n. Denn es ist:

$$\binom{2k}{k} = \frac{(2k)!}{k! \cdot k!}$$

Mit der Stirlingschen Formel für die Fakultät

$$k! \approx \left(\frac{k}{e}\right)^k \cdot \sqrt{2\cdot\pi\cdot k}$$

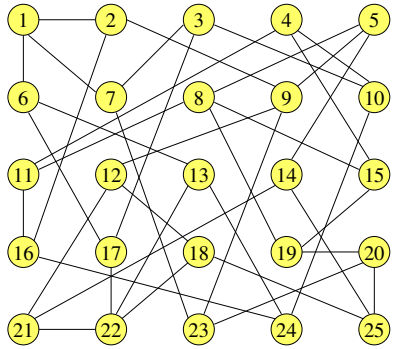
mit e = 2,71828182845904...
und π = 3,14159265358979...

folgt hieraus durch Einsetzen und Ausrechnen:

$$\binom{2k}{k} \approx \frac{4^k}{\sqrt{\pi \cdot k}}$$

mit $k^2 = n$.

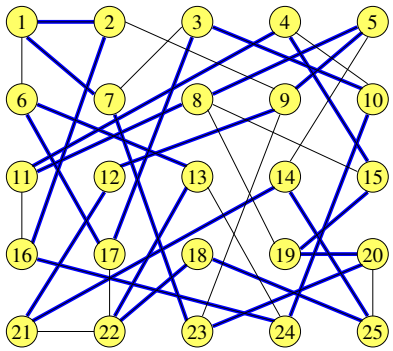
zu 2. Finde einen Weg, der jeden Knoten genau einmal besucht



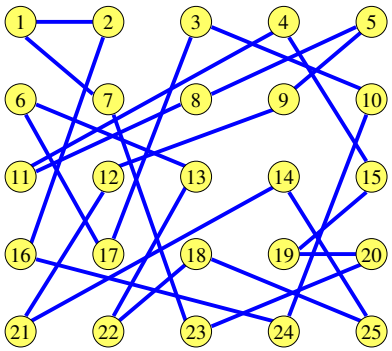
Knoten: 25

Kanten: 39

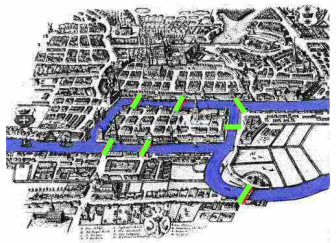
Gibt es in diesem Graphen einen Hamiltonschen Kreis? **Ja.**



Kreis: 1, 2, 16, 24, 10, 3, 17, 6, 13, 22, 18, 25, 14, 21, 12, 9, 5, 8, 11, 4, 15, 19, 20, 23, 7, 1

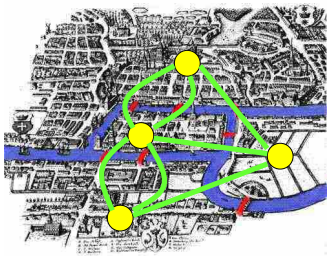


zu 3. Eulersche Kreise:
Das Königsberger Brückenproblem (1736)

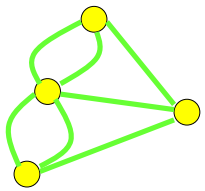


Kann man einen Rundgang durch Königsberg machen, so dass man jede der 7 Brücken genau einmal überquert und am Ende wieder am Startpunkt ankommt? Dieses Problem gilt als der Beginn der Graphentheorie.

Konstruiere hierzu einen Graphen (mit Mehrfachkanten)

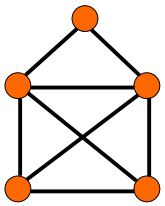


Satz: Ein zusammenhängender ungerichteter Graph besitzt genau dann einen Eulerschen Kreis, wenn alle seine Knoten einen geraden Grad haben.



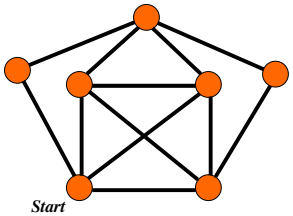
Eulers Beweis ergab: Es gibt keinen Rundgang durch Königsberg, auf dem jede Brücke genau einmal benutzt wird.

Zusatz: Haben genau zwei Knoten einen ungeraden Grad, so hat der Graph einen Eulerschen Weg, aber keinen Eulerschen Kreis.



Beispiel: "Dies ist das Haus des Nikolaus".

Durch Hinzunahme eines oder zweier Knoten mit geeigneten Kanten erhalten wir hieraus einen Graphen mit einem Eulerschen Kreis.



8.8.9 Durchsuchen eines Graphens

Gesucht ist also ein Algorithmus, der an irgendeinem Knoten u beginnt und dann alle von u aus erreichbaren Knoten und Kanten besucht.
Im ungerichteten Fall wird hierbei genau die zu u gehörige Zusammenhangskomponente G(u) = (Z(u), E'(u)) durchlaufen, siehe 3.8.9. Im gerichteten Fall wird der von u aus erreichbare Teil der schwachen Zusammenhangskomponente (speziell: alle Knoten v mit (u,v)∈ E*, siehe transitive Hülle) besucht. Werden auf diese Weise nicht alle Knoten des Graphens erreicht, so muss man dieses Vorgehen mit irgendeinem bisher noch nicht besuchten Knoten fortsetzen. Vergleiche hierzu 3.8.11 und 12.

Wir fassen das bereits Bekannte zusammen und ergänzen es. Zum Durchlauf von Bäumen siehe Abschnitt 3.7.6.

Ausgehend von einem Knoten u werden die Knoten und Kanten meist nach zwei Strategien aufgesucht:

Tiefensuche (**DFS** = depth first search): Erreicht man einen Knoten v, so wird ab hier rekursiv weiter gesucht, indem man allen von v ausgehenden Kanten folgt. Stößt man hierbei auf einen bereits besuchten Knoten, so wird in dieser Richtung nicht weiter gesucht (Abbruch der Rekursion).

Breitensuche (**BFS** = breadth first search): Man durchläuft den Graphen, ausgehend von u, schalenförmig gemäß des Abstandes, d.h., man besucht zunächst alle Knoten v mit dem Abstand dist(u,v) = 1, dann alle Knoten v mit dem Abstand dist(u,v) = 2 usw.

Tiefensuche (**DFS** = depth first search) umgangssprachlich.

Gerichteter Fall :

```
procedure besuche(u) is
begin  "bearbeite den Knoten u;"  markiere u als besucht;
       for alle v aus der Menge der Nachfolger S(u) loop
           if v noch nicht besucht then besuche(v); end if;
       end loop;
end besuche;
```

Im ungerichteten Fall ersetzt man die Menge S(u) durch die Menge der Nachbarn N(u), siehe 3.8.5e.

Wir betrachten im Folgenden nur den gerichteten Fall. Die Tiefensuche läuft im Prinzip wie im Algorithmus GD aus 3.8.7 ab. Zur Datenstruktur siehe oben bzw. 3.8.6.

Rekursives Ada-Programm zur *Tiefensuche*, gerichteter Fall.

```
procedure DFS (Anfang: in NextKnoten) is
p: NextKnoten;                    -- Datentypen siehe am Ende von 3.8.6
procedure besuche (u: in NextKnoten) is
    edge: NextKante; v: NextKnoten;
    begin  u.Besucht := true; edge := u.EIK;
        while  edge /= null  loop  v := edge.EKn;
            if not v.Besucht then besuche(v); end if;
            edge := edge.NKa;  end loop;
    end besuche;
begin p := Anfang;
    while p /= null loop  p.Besucht := false;  p:=p.NKn;  end loop;
    p := Anfang;
    while p /= null loop  if not p.Besucht then besuche(p); end if;
        p := p.NKn;  end loop;
end DFS;
```


Iteratives Programm zur *Tiefensuche*, gerichteter Fall. Zu den Kellern vgl. Abschnitte 4.3.3 und 4.4.3 (generisches Paket "Stack" sei hier sichtbar).

```
procedure DFS (Anfang: in NextKnoten) is
p, v: NextKnoten; KK: new Stack (item => NextKnoten); edge: NextKante;
begin p := Anfang;
    while p /= null loop  p.Besucht := false; p:=p.NKn;  end loop;
    p := Anfang ;
    while p /= null loop
    if not p.Besucht then Push(p,KK);
        while not Isempty(KK) loop
            u := Top(KK); Pop(KK); u.Besucht := true; edge := u.EIK;
            while  edge /= null  loop  v := edge.EKn;
                if not v.Besucht then push(v,KK); end if;
                edge := edge.NKa;
            end loop;
        end loop;
    end if;
    p := p.NKn;
    end loop:
end DFS;
```


Iteratives Programm zur Breitensuche, gerichteter Fall: Wie Tiefensuche, aber ersetze den Keller durch eine Schlange! (blau = Unterschied zu DFS)

```
procedure BFS (Anfang: in NextKnoten) is
p, v: NextKnoten; "KK: new Schlange (NextKnoten);" edge: NextKante;
begin p := Anfang;
    while p /= null loop  p.Besucht := false; p:=p.NKn;  end loop;
    p := Anfang ;
    while p /= null loop
    if not p.Besucht then Enter (p,KK);
        while not Isempty(KK) loop
            u := First(KK); Remove(KK); u.Besucht := true; edge := u.EIK;
            while  edge /= null  loop  v := edge.EKn;
                if not v.Besucht then Enter(v,KK); end if;
                edge := edge.NKa;
            end loop;
        end loop;
    end if;
    p := p.NKn;
    end loop:
end BFS;
```


Tiefensuche mit der Adjazenzmatrix A als Darstellung:

```
procedure DFS_Adj is                          -- n ist hier global
A: array (0..n-1, 0..n-1) of Integer;
Besucht: array(0..n-1) of Boolean;
procedure besuche (j: in 0..n-1) is
    begin  Besucht(j) := true;
        for k in 0..n-1 loop
            if A(j,k) /= 0 and not Besucht(k)
            then besuche (k); end if;
        end loop;
    end besuche;
begin  ...          -- die Adjazenzmatrix A möge aufgebaut worden sein
    for i in 0..n-1 loop  Besucht(i) := false;  end loop;
    for i in 0..n-1 loop
        if not Besucht(i) then besuche (i); end if; end loop;
end DFS_Adj;
```


Zur Zeitkomplexität:

Adjazenzlistendarstellung: Jeder Durchlauf besucht jede Kante genau einmal. Jeder Knoten wird so oft besucht, wie Kanten auf ihn verweisen, mindestens aber einmal. Also ist die Zeitkomplexität linear O(n+m).

Adjazenzmatrix: Da eine Matrix angelegt werden muss, beträgt der Aufbau des Graphens $\Theta(n^2)$ Schritte, auch wenn er nur relativ wenige Kanten besitzt. In der Prozedur wird bei jedem Aufruf von “besuche” die for-k-Schleife n Mal durchlaufen; weil “besuche” mindestens n Mal aufgerufen werden muss, ergibt sich eine Zeitkomplexität von $O(n^2)$.

Überlegen Sie, wie viele Schritte die Verfahren für einen vollständigen Graphen K_n und für einen Graphen mit n isolierten Knoten (also ein Graph ohne Kanten) benötigen.

8.9 Sonstiges

8.9.1: Anregung für analytisch Interessierte

Wie kann man einer Rekursionsgleichung ansehen, welche Lösungen sie hat? (siehe Herleitung von Satz 8.2.15)

Das lässt sich nicht pauschal beantworten. Denn schließlich ist dieses Problem nicht entscheidbar. Manchmal reicht es aber bereits, wenn man die Gleichung umformt (erweitern, einsetzen, in irgendeinem Sinne vereinfachen usw.).

Oft ist es hilfreich, die "Ableitung" zu betrachten, also F(n) - F(n-1), oder die zweite Ableitung
(F(n) - F(n-1)) - (F(n-1) - F(n-2)) = F(n) - 2 F(n-1) + F(n-2).

Dies liefert einen Hinweis, wie die Lösung aussehen könnte, und man kann dann mit einem Lösungansatz, den man in die Gleichung einsetzt, versuchen, eine Lösung aufzuspüren.

Wir betrachten als Beispiel die Rekursionsformel für F(n):

$F(n) = (n+1)/n \cdot F(n-1) + (2n-1)/n$. Einfaches Umformen liefert:
$F(n) = F(n-1) + 1/n \cdot F(n-1) + (2n-1)/n$, d.h.:
$F(n) - F(n-1) = 1/n \cdot F(n-1) + (2n-1)/n$.
Hier sieht man wenig, weil der Wert F(n-1) noch auf der rechten Seite stehen geblieben ist.
Daher versuchen wir nun, die zweite Ableitung zu berechnen.
Aus obiger Formel für F(n)-F(n-1) folgt:
$F(n-1) - F(n-2) = 1/(n-1) \cdot F(n-2) + (2n-3)/(n-1)$.
Man subtrahiere die letzte Formel von der davor:
$(F(n) - F(n-1)) - (F(n-1) - F(n-2))$
$= 1/n \cdot F(n-1) + (2n-1)/n - 1/(n-1) \cdot F(n-2) - (2n-3)/(n-1)$
$= 1/n \cdot F(n-1) - 1/(n-1) \cdot F(n-2) + 1/(n \cdot (n-1))$.
$1/n \cdot F(n-1) - 1/(n-1) \cdot F(n-2)$ erinnert nun an die Rekursionsformel, denn dort steht (ersetze n durch n+1):
$1/(n+1) \cdot F(n) = 1/n \cdot F(n-1) + \ldots$

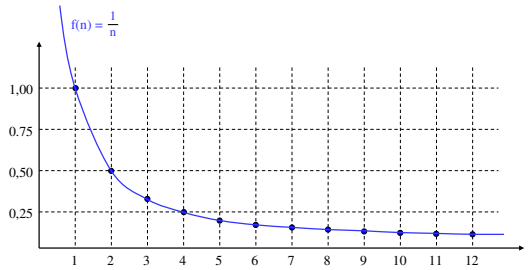
Also gilt:
$1/n \cdot F(n-1) = 1/(n-1) \cdot F(n-2) + (2n-3)/(n \cdot (n-1))$ bzw.
$1/n \cdot F(n-1) - 1/(n-1) \cdot F(n-2) = (2n-3)/(n \cdot (n-1))$.
Dies setzt man oben ein:
$(F(n) - F(n-1)) - (F(n-1) - F(n-2))$
$= 1/n \cdot F(n-1) - 1/(n-1) \cdot F(n-2) + 1/(n \cdot (n-1))$
$= (2n-3)/(n \cdot (n-1)) + 1/(n \cdot (n-1))$
$= (2n-2)/(n \cdot (n-1)) = 2/n$

Es entsteht ein überraschend einfacher Ausdruck. Nun erinnern wir uns an die Stammfunktionen aus der Analysis: 1/n ist die Ableitung des Logarithmus ln(n). Also müsste die "Ableitung" ln(n) und damit die Funktion F von der Form $n \cdot \ln(n)$ sein. Da keine kontinuierliche Funktion herauskommen wird, sollte man H(n) anstelle von ln(n) nehmen, d.h., das Ergebnis könnte die Form $F(n) = a \cdot n \cdot H(n) + b \cdot n + c$ (mit Konstanten a, b, c) haben.

Dies setzt man nun in die ursprüngliche Gleichung
$F(n) = (n+1)/n \cdot F(n-1) + (2n-1)/n$ ein:
$a \cdot n \cdot H(n) + b \cdot n + c$
$= (n+1)/n \cdot (a \cdot (n-1) \cdot H(n-1) + b \cdot (n-1) + c) + (2n-1)/n$
Multiplikation mit n liefert:
$a \cdot n^2 \cdot H(n) + b \cdot n^2 + c \cdot n$
$= (n+1) \cdot a \cdot (n-1) \cdot H(n-1) + b \cdot (n^2-1) + c \cdot (n+1) + (2n-1)$, also
$a \cdot n^2 \cdot H(n-1) + a \cdot n = a \cdot (n^2-1) \cdot H(n-1) - b + c + (2n-1)$
$a \cdot H(n-1) + a \cdot n = c - b + (2n-1)$.
Man sieht, dass diese Gleichung für Konstanten a, b und c nicht zu erfüllen ist. Man braucht offenbar im Ansatz ein weiteres Glied $d \cdot H(n)$. Neuer Ansatz:
$F(n) = a \cdot n \cdot H(n) + b \cdot n + c + d \cdot H(n)$.
Wegen der Nebenbedingung F(0)=0 muss c=0 sein. Erneut einsetzen:

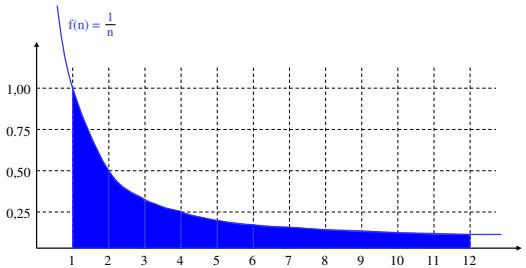
$a \cdot n \cdot H(n) + b \cdot n + d \cdot H(n)$
$= (n+1)/n \cdot (a \cdot (n-1) \cdot H(n-1) + b \cdot (n-1) + d \cdot H(n-1)) + (2n-1)/n$
Multiplikation mit n liefert:
$a \cdot n^2 \cdot H(n) + b \cdot n^2 + d \cdot n \cdot H(n)$
$= a \cdot n^2 \cdot H(n-1) + a \cdot n + b \cdot n^2 + d \cdot n \cdot H(n)$
$= (n+1) \cdot a \cdot (n-1) \cdot H(n-1) + b \cdot (n^2-1) + d \cdot (n+1) \cdot H(n-1) + (2n-1)$
$= a \cdot (n^2-1) \cdot H(n-1) + b \cdot (n^2-1) + d \cdot n \cdot H(n-1) + d \cdot H(n-1) + (2n-1)$.
Umformen:
$a \cdot H(n-1) + a \cdot n + b \cdot n^2 + d \cdot n \cdot H(n)$
$= b \cdot (n^2-1) + d \cdot n \cdot H(n-1)) + d \cdot H(n-1)) + (2n-1)$ wird zu
$a \cdot H(n-1) + a \cdot n + d = -b + d \cdot H(n-1) + (2n-1)$.
Wenn dies lösbar ist, so muss a = d und a = 2 sein; es verbleibt:
$2 = -b - 1$, d.h., b = -3. Somit haben wir eine Lösung gefunden:
$F(n) = 2 \cdot n \cdot H(n) - 3 \cdot n + 2 \cdot H(n)$, also die gleiche Lösung wie im Beweis zu Satz 8.2.15.

8.9.2: Veranschaulichung der harmonischen Funktion 8.2.14



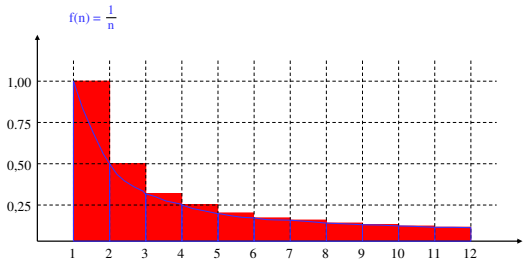
Eine ausführlichere Untersuchung steht in Abschnitt 1.5.

Abschätzung der harmonischen Funktion



Der natürliche Logarithmus ln(n) ist die Fläche unterhalb der Kurve von 1 bis n.

Abschätzung der harmonischen Funktion



Der natürliche Logarithmus ln(n) ist die Fläche unterhalb der Kurve von 1 bis n.
Die harmonische Funktion H(n-1) ist durch die rote Fläche von 1 bis n gegeben.
Man sieht: $H(n) - 1 \le \ln(n) \le H(n-1)$ für alle $n \ge 1$. Die kleinen über der blauen Kurve liegenden roten Flächenstücke summieren sich zur Eulerschen Konstanten $\gamma = 0{,}5772156649\ldots$ auf.