UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

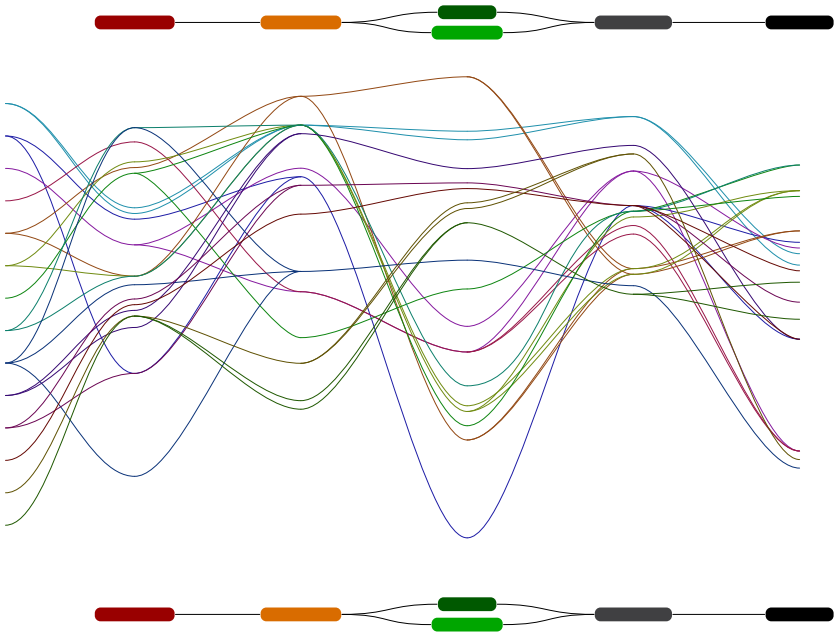
Vorlesungsskript

Algorithmendesign

CS3000, Wintersemester 2016/2017

Fassung vom 18. November 2016

Till Tantau

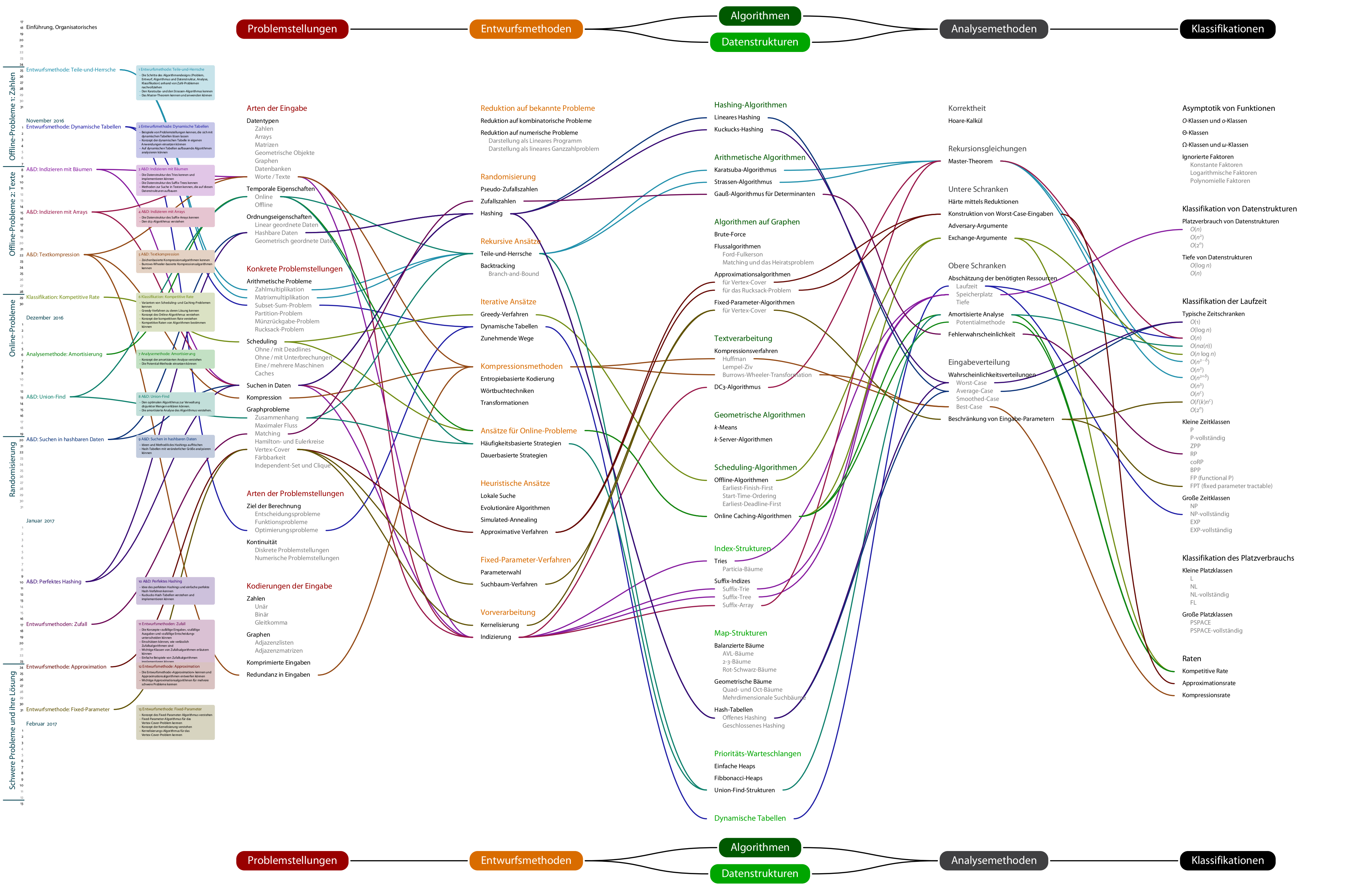


UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR THEORETISCHE INFORMATIK

Veranstaltungskarte
CS3000 Algorithmendesign

ALGORITHMENDESIGN

TILL TANTAU
WWW.TCS.UNI-LUEBECK.DE
11. OKTOBER 2016



Inhaltsverzeichnis

Teil I
Offline-Probleme 1: Zahlen

1 Entwurfsmethode: Teile-und-Herrsche

2 Entwurfsmethode: Dynamische Tabellen

Teil II
Offline-Probleme 2: Texte

3 A&D: Indizieren mit Bäumen

4 A&D: Indizieren mit Arrays

Teil V

Entwurfsmethoden für schwere Probleme

Vorwort

Sie fangen als frisch gebackener Informatiker in einer Firma für Sensornetztechnik an und schon nach wenigen Tagen kommt Ihre jung-dynamische Chefin an und lädt folgendes Problem auf Ihrem Schreibtisch ab: »Der Kunde hat uns gebeten, noch Bewegungssensoren für das Gebäude zu einzuplanen, so dass jeder Quadratzentimeter Korridor von mindestens einem Sensor erfasst wird. Ich schicke dir gleich noch die Datei mit den Gebäudeplänen. Ach ja, es sollen natürlich möglichst wenige Sensoren verbaut werden, die Dinger kosten ein Heidengeld. Schaffst du das bis morgen?« Bevor Sie verneinen können, ist sie auch schon wieder weg.

In dieser Veranstaltung werden Sie lernen, wie Sie ein solches Problem angehen können. Algorithmendesign dreht sich genau darum, ein unbekanntes Problem »anzugehen«, um am Ende einen Algorithmus (und zugehörige Datenstrukturen) zu dessen Lösung zu bekommen. Die verschiedenen Arten, wie man ein Problem »angehen« kann, nennt man vornehmer *Entwurfsmethoden* oder ganz genau *Algorithmen-Entwurfsmethoden*, um sie von den Methoden der Softwaretechnik zum Entwurf von großen Softwaresystemen etwas abzugrenzen. Das wohl bekanntestes Beispiel einer solchen Entwurfsmethode ist das Teile-und-Herrsche-Prinzip, das Sie schon kennen (sollten), andere Beispiele sind Reduktionstechniken, Approximationsmethoden, Heuristiken, Randomisierung oder auch Fixed-Parameter-Methoden. (Wenn Ihnen diese Begriffe jetzt noch nichts sagen, dann ist das gut so, sonst könnten Sie sich die Veranstaltung auch sparen.)

Mit dem Entwurf und sogar mit der Implementation eines neuen Algorithmus ist es aber nicht getan: Die *Analyse* von Algorithmen ist (fast) genauso wichtig wie deren Entwicklung. Der Grund ist ganz einfach: Was nützt mir der raffinierteste Algorithmus, wenn er bei meinen Eingaben nicht zu meinen Lebzeiten fertig wird? Schlimmer noch: Selbst wenn er bei *meinen* Eingaben in der Praxis schnell ist, was passiert, wenn er bei den Eingaben des *Kunden* plötzlich versagt? Ein klassisches Beispiel ist das Programm Word von Microsoft: Es eignet sich hervorragend, einen Brief von zwei Seiten zu schreiben. Wer aber schon einmal versucht hat, eine Arbeit von mehreren hundert Seiten damit zu bearbeiten, weiß, dass bei Microsoft offenbar noch nie jemand so lange Texte geschrieben hat (oder Microsoft benutzt selbst gar nicht Word, was einiges erklären würde).

Das zentrale Ziel der Analyse ist es, das Verhalten von Algorithmen bei beliebigen (oder zumindest bei sehr wahrscheinlichen) Eingaben vorherzusagen. Diesen Aspekt der *Algorithmen-Analyse* werden wir in dieser Veranstaltung auch eingehend betrachten. Wieder kennen Sie bereits Verfahren aus früheren Veranstaltungen, beispielsweise die Ermittlung der O-Klasse der Laufzeit eines Algorithmus. Und wieder werden Sie neue Verfahren kennen lernen, mit denen sich leichter oder besser Aussagen über das Verhalten von Algorithmen und Datenstrukturen machen lassen. Solche »Aussagen über das Verhalten« nennen wir dann *Klassifikationen,* das letzte Glied in der Kette *Problem – Entwurf – Algorithmus / Datenstruktur – Analyse – Klassifikation*.

Im Rahmen dieser Veranstaltung werden wie die Kette für immer neue Probleme immer wieder neu durchlaufen. Wir werden exemplarisch Probleme der unterschiedlichsten Arten betrachten: Probleme auf Zahlen, Probleme auf Texten, Problem auf Graphen, Probleme auf geometrischen Objekten, Probleme, bei denen die Eingaben am Anfang gar nicht vollständig vorliegen, oder auch Probleme, bei denen manche Eingaben wahrscheinlicher sind als andere. Der Fokus wird oft, aber nicht immer, auf der Lösung der Probleme liegen; manchmal geht es eher darum, eine Entwurfs- oder Analysemethode genauer zu betrachten.

Jedes Kapitel dieses Skripts entspricht grob einer Vorlesungsdoppelstunde und mit jedem Kapitel verfolge ich gewisse Ziele, welche Sie am Anfang des jeweiligen Kapitels genannt finden. Neben diesen etwas kleinteiligen Zielen gibt es auch folgende zentralen »offiziellen« Veranstaltungsziele, wie sie auch im Modulhandbuch zu finden sind:

1. Vertrautheit mit algorithmischen Entwurfsprinzipien.
2. Neue komplexe Algorithmen durch Anwendung dieser Prinzipien entwickeln können.
3. Erfahrung beim algorithmischen Problemlösen.

Das Wörtchen »können« taucht bei den Veranstaltungszielen mehrfach auf. Um etwas wirklich zu können, reicht es nicht, davon gehört zu haben oder davon gelesen zu haben. Man muss es auch wirklich *getan* haben: Sie können sich tausend Pokerspiele im Fernsehen anschauen, sie sind deshalb noch nicht mit Poker reich werden; sie können tausend Stunden World of Warcraft spielen, sie werden deshalb trotzdem keine Feuerkugeln auf Ihren Professor geschleudert bekommen.

Deshalb steht bei dieser Veranstaltung der Übungsbetrieb mindestens gleichberechtigt neben der Vorlesung. Der Ablauf ist dabei folgender: In der Vorlesung werde ich Ihnen die Thematik vorstellen und Sie können schon mit dem Üben im Rahmen kleiner Miniübungen *während der Vorlesung* beginnen. Alle zwei Wochen gibt es ein Übungsblatt, das inhaltlich zu den Vorlesung gehört. Sie müssen sich die Übungsblätter aber nicht »alleine erkämpfen«. Vielmehr gibt es Tutorien, in denen Sie Aufgaben üben werden, die »so ähnlich« wie die Aufgaben auf den Übungsblättern sind. Sie werden feststellen, dass die als »leicht« und in der Regel auch die als »mittel« eingestuften Aufgaben mit der Vorbereitung im Tutorium in der Tat mit vertretbarem Aufwand schaffbar sind. Ist eine Aufgabe »schwer«, so ist es kein Unglück, wenn Sie diese nicht schaffen – probieren sollten Sie es aber trotzdem.

Ich wünsche Ihnen viel Spaß mit dieser Veranstaltung.

Till Tantau

Teil I

Offline-Probleme 1: Zahlen

Womit ließe sich trefflicher eine Veranstaltung über Algorithmen beginnen als mit der vornehmsten Aufgabe eines jeden Computers, dem *Rechnen*? Auch wenn man es heute kaum noch glauben mag: Computer sind ursprünglich *nicht* dafür entworfen worden, unser Leben in Timelines zu dokumentieren, sondern ganz banal für das Rechnen – wie ja der Name »Computer« auch schon nahelegt.

Die Welt der Zahl-Probleme ist vielfältig und reich, wir werden nur einen sehr kleinen Ausschnitt betrachten können. Beginnen möchte ich mit der Multiplikation, allerdings nicht nur der von einfachen Zahlen, sondern auch gleich von ganzen Matrizen. Man sollte meinen, dass dieses Problem wie man so schön sagt »wohl untersucht« sein sollte, jedoch ist es völlig offen, wie schnell man zwei Matrizen nun wirklich multiplizieren kann; es gibt also selbst bei den scheinbar einfachen Problemen noch viel zu tun für angehende Algorithmendesignerinnen und -designer. Wir werden uns dann auch an nachweisbar schwere Probleme machen wie das Subset-Sum-Problem und diese mit Hilfe von dynamischen Tabellen angehen.

In diesem, wie in allen folgenden Teilen, stehen die eigentlichen Rechen-Probleme nicht immer im Vordergrund. Vielmehr geht es um den Prozess, wie man von einer Problemstellung, und sei es so eine »einfache« wie die Multiplikation von zwei Zahlen, zu einem korrekten, effizienten Algorithmus kommt.

1-1

Kapitel 1

Entwurfsmethode: Teile-und-Herrsche

»Gaussian Elimination is not Optimal«

1-2

Lernziele dieses Kapitels

1. Die Schritte des Algorithmendesigns (Problem, Entwurf, Algorithmus und Datenstruktur, Analyse, Klassifikation) anhand von Zahl-Problemen nachvollziehen
2. Den Karatsuba- und den Strassen-Algorithmus kennen
3. Das Master-Theorem kennen und anwenden können

Inhalte dieses Kapitels

Worum es heute geht

Author: David Eppstein, Creative Commons Attribution Licence, Volker Strassen gibt die 2008 Knuth-Preis-Vorlesung beim 20th ACM-SIAM Symposium on Discrete Algorithms

Wissenschaft ist immer dann am spannendsten, wenn scheinbar offensichtliche Wahrheiten über den Haufen gestoßen werden. Eine solche offensichtliche Wahrheit war, dass man zwei Matrizen, jede der Größe $n \times n$, nicht schneller als in Zeit $O(n^3)$ multiplizieren könne. In der Tat: Die Produktmatrix besteht hat n^2 viele Einträge und jeder dieser Einträge ist die Summe von n Produkten von je zwei Zahlen in den Eingabematrizen. Es ist schwer vorstellbar, wie man schneller als in Zeit n^3 gerade n^2 Mal eine Summer von je n Zahlen bestimmen soll.

Es war deshalb ein ziemlicher Paukenschlag, als Volker Strassen 1968 ein dreiseitiges Papier bei der Zeitschrift *Numerische Mathematik* einreichte mit dem schlichten Titel »Gaussian Elimination is not Optimal«, in dem gezeigt wird, wie man zwei Matrizen *schneller* als in Zeit n^3 multipliziert. Die entscheidende Idee dabei ist, einen *Teile-und-Herrsche-Ansatz* zu verfolgen, wobei man sich allerdings recht geschickt anstellen muss; einfach drauf-los-teilen-und-herrschen führt nicht zum Ziel.

Es ist nicht sonderlich schwierig einzusehen, dass Strassens Algorithmus korrekt arbeitet. Schwieriger ist es hingegen, seine Laufzeit zu analysieren: Der Algorithmus schafft es, in jeder Rekursion nur sieben statt acht rekursive Aufrufe zu machen – was heißt das nun aber für die Laufzeit? Da solche Fragen häufiger auftauchen, werden wir uns mit Hilfe des so genannten Master-Theorems ein Werkzeug zurechtlegen, mit dem sie leicht beantwortet werden können.

Seit Strassens Beitrag ist die Laufzeit für Algorithmen zur Matrixmultiplikation immer weiter verbessert worden, weshalb mittlerweile die Vermutung kursiert, dass man Matrizen sogar in Zeit $O(n^2)$ multiplizieren kann; zeigen konnte dies allerdings noch niemand.

1.1 Was ist Algorithmendesign?

Zur Einstimmung: ein Zahlproblem aus Theorie *und* Praxis

1-4

Das Primzahlproblem

Eingabe Eine ganze Zahl in Binärdarstellung mit n Bits.

Frage Ist die Zahl prim? (Hat sie genau zwei Teiler?)

Dieses Problem ist uralt, es wird bereits bei Euklid behandelt.

Papyrus von Euklid

Seine Lösung wird in gängigen Public-Key-Kryptosystemen eingesetzt.

Offizielles Werk des Bundes.

✎ Zur Diskussion

Wie würden Sie das Problem lösen? Wie schnell wäre Ihre Lösung bei Eingaben mit 1024 Bit? Ist das in der Praxis schnell genug?

Zum Vergleich: Der Miller-Rabin-Test beantwortet die Frage innerhalb weniger Millisekunden.

Worum geht es beim Algorithmendesign?

1-5

Algorithmendesign bietet *Methoden*, um *Probleme* systematisch zu lösen:

1. Ausgangspunkt ist eine *Problemstellung*.
   Beispiel: Primzahltest
   Heute: Multiplikation von Zahlen und Matrizen.
2. Auf diese wendet man eine *Entwurfsmethode* an.
   Beispiel: Randomisierung
   Heute: Teile-und-Herrsche
3. Im Ergebnis erhält man *Algorithmen* und *Datenstrukturen*.
   Beispiel: Miller-Rabin-Test
   Heute: Karatsuba- und Strassen-Algorithmen
4. Die »Qualität« der Lösung bestimmt man mittels *Analysemethoden*.
   Beispiel: Average-Case-Analyse
   Heute: Rekursionsgleichungen und Master-Theorem
5. Im Ergebnis erhält man eine *Klassifikation*.
   Beispiel: Erwartete polynomielle Laufzeit
   Heute: O-Klassen

1.2 Wiederholung: O-Klassen, Ω-Klassen und Θ-Klassen

1-6

Wiederholung: Wie kann man die Rechenzeit von Algorithmen vergleichen?

Sollte man lieber Insertion-Sort oder Merge-Sort benutzen? Laufzeitmessungen ergeben folgende Werte:

n zufällige Zahlen	Insertion-Sort	Merge-Sort
$n = 3$	$0{,}056\mu s$	$0{,}183\mu s$
$n = 10$	$0{,}360\mu s$	$1{,}080\mu s$
$n = 1000$	$1.250\mu s$	$190\mu s$
$n = 100.000$	$12.122.000\mu s$	$35.000\mu s$

Moral

1. Ein guter Algorithmus schlägt einen schlechten Algorithmus, selbst wenn der gute Algorithmus schlecht implementiert wird und der schlechte gut implementiert wird.
2. Die Terme n und $\log n$ in Laufzeiten sind *in der Regel* wichtiger
   - als die verwendete Hardware,
   - als das Geschick der Programmierer.

1-7

O-Klassen = höchstens.

▶ **Definition:** Groß-O-Klasse

Sei $g\colon \mathbb{N} \to \mathbb{R}_0^+$ eine Funktion. Dann ist die O-Klasse $O(g)$ die Menge aller Funktionen $f\colon \mathbb{N} \to \mathbb{R}_0^+$, für die

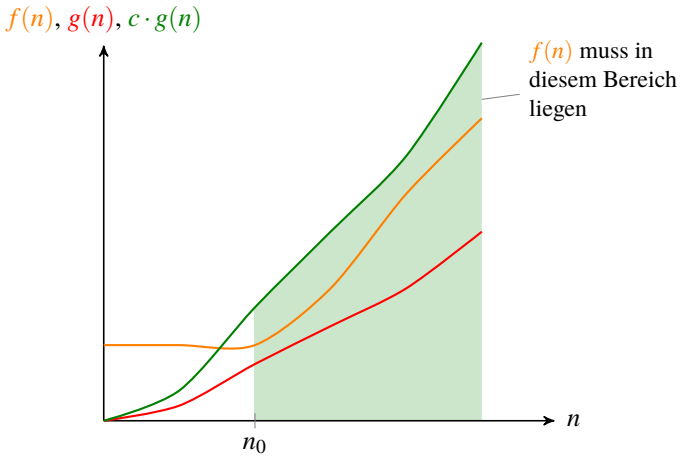
- es eine Konstante c gibt und
- es eine Konstante n_0 gibt, so dass
- für alle $n > n_0$ gilt $f(n) \leq c \cdot g(n)$.

Merke

»$f \in O(g)$« bedeutet, dass für *genügend große* n und einen *genügend großen Faktor* c gilt, dass $c \cdot g(n)$ immer größer als $f(n)$ ist.

1-8

Veranschaulichung von $f \in O(g)$



1-9

Beispiele zu O-Klassen.

Beispiel

Sei $f(n) = 3 + 17n^2$ und $g(n) = n^2$. Dann ist $f \in O(g)$.
(Wähle $c = 1000$ und $n_0 = 1000$. Dann ist sicherlich $3 + 17n^2 \leq cn^2$.)

Beispiel

Sei $g(n) = 1$ für alle n. Dann enthält die Klasse $O(g)$ alle beschränkten Funktionen.

Beispiel
Sei $f(n) = n^2$ und $g(n) = n$. Dann ist $f \notin O(g)$.

Beispiel
Sei $f(n) = \log_2(n^2)$ und $g(n) = \log_2 n$. Dann ist $f \in O(g)$.
(Es gilt $\log_2(n^2) = 2\log_2 n$.)

Zur Schreibweise von O-Klassen.

1-10

- Man schreibt einfach »$O(n^2)$« für »$O(g)$, wobei g die Funktion $g(n) = n^2$ ist«.
- Man schreibt auch »$f(n) = O(n^2)$« statt »$f \in O(n^2)$«.
- Man schreibt auch Dinge wie »$2^{O(n)}$« und meint damit eigentlich »Eine Funktion, die für hinreichend große n und eine hinreichend große Konstante c durch $2^{c \cdot n}$ nach oben beschränkt ist.«
- *Das ist mathematisch alles Quatsch – aber das stört niemanden.*

O-Klassen = höchstens. Ω-Klassen = mindestens. Θ-Klassen = genau.

1-11

- Eine O-Klasse enthält alle Funktionen, die *höchstens* soundso schnell wachsen.
- Eine Ω-Klasse enthält alle Funktionen, die *mindestens* soundso schnell wachsen.
- Eine Θ-Klasse enthält alle Funktionen, die *genau* soundso schnell wachsen.

▶ Definition: Groß-Omega-Klassen
Es gilt $f \in \Omega(g)$ genau dann, wenn $g \in O(f)$.

▶ Definition: Theta-Klassen
$\Theta(g) = O(g) \cap \Omega(g)$.

Beispiel: Maximumsbestimmung
Ein Programm soll das Maximum einer Liste von Zahlen bestimmen. Dann muss es *jede Zahl mindestens einmal untersuchen* (sonst könnte man nämlich eine Eingabe bauen, bei der das Programm eine falsche Ausgabe macht). Der Aufwand des Programms ist also *mindestens linear*, das heißt $\Omega(n)$. Da er andererseits auch *höchstens linear* ist, liegt der Zeitaufwand sogar in $\Theta(n)$.

Beispiel: Sortieren
Ein Programm soll eine Liste von Objekten sortieren. Man kann zeigen, dass man hierzu mindestens $n \log_2 \frac{n}{e}$ Vergleiche benötigt. Der Aufwand eines solchen Programms ist also $\Omega(n \log n)$.

✎ Zur Übung

1-12

Geben Sie für folgende f und g an, ob $f \in O(g)$, ob $f \in \Omega(g)$ und ob $f \in \Theta(g)$ gelten:

1. $f(n) = \sqrt{n}$ und $g(n) = n$.
2. $f(n) = n^{4,00001}$ und $g(n) = n^4$.
3. $f(n) = n^5$ und $g(n) = n^4(\log n)^7$.
4. $f(n) = \log_2 n$ und $g(n) = 1$.
5. $f(n) = \log_3 n$ und $g(n) = \log_2 n$.
6. $f(n) = 2^n$ und $g(n) = 3^n$.

Für noch mehr Klassen und eine ausführlichere Einführung siehe das Kapitel zur O-Notation im Skript zur Theoretischen Informatik 2009.

1.3 Die Schritte des Algorithmendesigns

1.3.1 Die Probleme: Multiplikation

1-13

Zwei scheinbar einfache Problemstellungen.

Das Multiplikationsproblem

Eingabe Zwei Zahlen x und y in Binärdarstellung mit je n Bits.

Ausgabe Die Binärdarstellung des Produkts xy.

Das Multiplikationsproblem für Matrizen

Eingabe Zwei $n \times n$ Matrizen X und Y mit Zahleinträgen (sinnvoll kodiert).

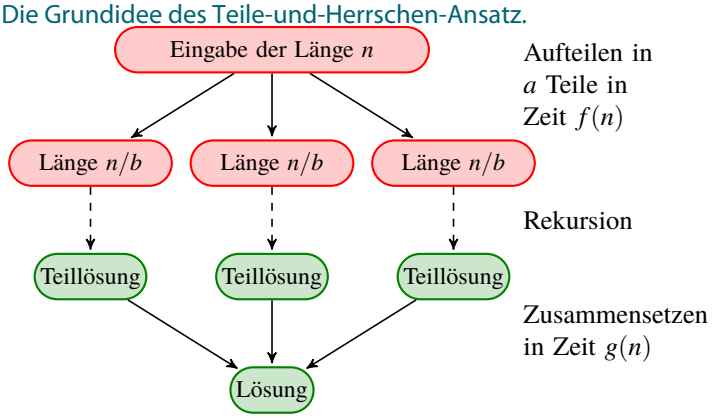
Ausgabe Das Matrix-Produkt XY (sinnvoll kodiert).

Bekanntermaßen gilt: Mittels der Schulmethode zur schriftlichen Multiplikation kann man zwei n-Bit-Zahlen in $\Theta(n^2)$ Zeitschritten multiplizieren. Mittels der Schulmethode zur Matrixmultiplikation kann man zwei $n \times n$ Matrizen mit $\Theta(n^3)$ Zahl-Multiplikationen multiplizieren.

1.3.2 Die Entwurfsmethode: Teilen-und-Herrschen

1-14

Die Grundidee des Teile-und-Herrschen-Ansatz.



1. Aus der Eingabe der Länge n werden kleinere Eingaben jeweils der Länge n/b errechnet. *Merke: »b wie Bruchteil«.*
2. Die Anzahl a der kleineren Eingaben ist oft gleich b, aber nicht immer. *Merke: »a wie Anzahl neuer Probleme«.*

1-15

✎ **Zur Übung**

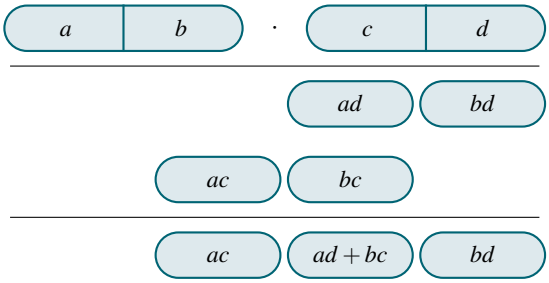
Wie lautet die Anzahl a der Teile, die Bruchzahl b, die Laufzeitschranke $f(n)$ für das Aufteilen und $g(n)$ für das Zusammensetzen für folgende Algorithmen:

1. Merge-Sort,
2. Quick-Sort,
3. Binäre Suche?

1.3.3 Die Algorithmen: Karatsuba und Strassen

Multiplikation per Teilen-und-Herrschen. 1-16
Erster Versuch.

Wir wollen zwei n-Bit-Zahlen x und y multiplizieren. Wendet man den *Teilen-und-Herrschen-Ansatz* an, so liegt es nahe, *jede Zahl in zwei Teile zu spalten.* Schreiben wir also x als $x = a \cdot 2^{n/2} + b$ und $y = c \cdot 2^{n/2} + d$, wobei a, b, c und d jeweils $n/2$ Bits haben. Das Produkt xy ist dann gleich $ac \cdot 2^n + (ad + bc)2^{n/2} + bd$. Statt x und y zu multiplizieren, können wir also die *vier* Produkte ac, ad, bc und bd berechnen.



✎ Zur Diskussion
Wie schnell ist der Algorithmus, der entsteht, wenn man wie oben angedeutet die vier Produkte ihrerseits jeweils rekursiv berechnet?

Multiplikation per Teilen-und-Herrschen. 1-17
Karatsubas Algorithmus.

Eine geniale Beoachtung
$ad + bc = (a+b)(c+d) - ac - bd$.

```
1  function Karatsuba(x,y)
2    n ← Anzahl Bits von x und y      // n muss eine Zweierpotenz sein
3    if n = 1 then
4      return x · y
5    else
6      a ← ersten n/2 Bits von x
7      b ← letzten n/2 Bits von x
8      c ← ersten n/2 Bits von y
9      d ← letzten n/2 Bits von y
10     ac ← Karatsuba(a,c)
11     bd ← Karatsuba(b,d)
12     z ← Karatsuba(a+b, c+d) − ac − bd
13     return ac · 2^n + z · 2^(n/2) + bd
```

Dieser Algorithmus kann zwei n-Bit-Zahlen mit *drei* rekursiven Aufrufen für $\frac{n}{2}$-Bit-Zahlen berechnen.

Matrix-Multiplikation per Teilen-und-Herrschen. 1-18
Erster Versuch.

Wir wollen nun zwei $n \times n$ Matrizen X und Y per Teilen-und-Herrschen multiplizieren. Es liegt nahe, diese Matrizen in *je vier Teile* aufzuspalten:

$$X = \begin{pmatrix} A & B \\ C & D \end{pmatrix},$$
$$Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}.$$

Dann gilt:

$$XY = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

Wir können so die Multiplikation von zwei $n \times n$ Matrizen auf die Multiplikation von *acht* $\frac{n}{2} \times \frac{n}{2}$ Matrizen rekursiv reduzieren.

1-19

Professor Howard und sein Daemon.

Kürzlich unterhielt sich Professor Howard vom Institute for Theoretical Computational Demonology (ITCD) mit seinem Daemon:

Howard Ich habe etwas über diesen Teile-und-Herrschen-Ansatz für das Matrizenproblem nachgedacht.

Daemon Ja, Meister. Und?

Howard Wir müssen doch vier Zahlen berechnen: $AE+BG$, $AF+BH$, $CE+DG$ und $CF+DH$.

Daemon Ja, Meister. Ihr seid weise, Meister.

Howard Ich frage mich nun, ob das auch mit nur sieben Multiplikationen geht.

Daemon Öh, Meister? Da sind acht Produkte zu berechnen.

Howard Ja, schon. Aber Karatsuba hat es auch geschafft, drei Zahlen, die vier Produkte enthalten, mit nur drei Multiplikationen zu berechnen.

Daemon Dem hat bestimmt sein Daemon geholfen, Meister.

Howard Gute Idee! Bei den Mächten der Informatik, beschwöre ich dich, mir eine Lösung für dieses Problem zu finden!

Die Mächte der Informatik ergreifen vom Daemon Besitz. Rauch steigt auf.

Daemon

$$
\begin{aligned}
P_1 &= (A+D)(E+H),\\
P_2 &= (C+D)E,\\
P_3 &= A(F+H),\\
P_4 &= D(G-E),\\
P_5 &= (A+B)H,\\
P_6 &= (C-A)(E+F),\\
P_7 &= (B-D)(G+H),\\
AE+BG &= P_1+P_4-P_5+P_7,\\
AF+BH &= P_3+P_5,\\
CE+DG &= P_2+P_4,\\
CF+DH &= P_1-P_2+P_3+P_6.
\end{aligned}
$$

Howard So ein Daemon ist schon praktisch…

1-20

Matrix-Multiplikation per Teilen-und-Herrschen.

Strassens Algorithmus.

```
1  function Strassen(X,Y)
2    // X und Y sind n × n Matrizen, n ist eine Zweierpotenz
3    if n = 1 then
4      return X · Y
5    else
6      A bis D ← die vier Quadranten von X
7      E bis F ← die vier Quadranten von Y
8      P1 ← Strassen(A + D, E + H)
9      ... // Gemäß den Angaben des Daemons
10     P7 ← Strassen(B − D, G + H)
11     return ( P1+P4−P5+P7   P3+P5
                P2+P4   P1−P2+P3+P6 )
```

Dieser Algorithmus kann zwei $n \times n$ Matrizen mit *sieben* rekursiven Aufrufen berechnen.

1.3.4 Die Analysemethode: Das Master-Theorem

Rekursionsgleichungen: Das Analyse-Werkzeug bei Teilen-und-Herrschen

1-21

Schritte beim Teilen-und-Herrschen bei Eingaben der Länge n:

1. Bilde in Zeit $f(n)$ irgendwie a neue Eingaben der Größe n/b.
2. Wende den Algorithmus auf jede dieser a Teileingaben rekursiv an.
3. Setze in Zeit $g(n)$ die Ergebnisse irgendwie zusammen.

Ist $T(n)$ die Laufzeit des Algorithmus bei Eingaben der Länge n, so ergibt sich folgende *Rekursionsgleichung* für die Laufzeit:

$$T(n) = f(n) + a \cdot T(n/b) + g(n).$$

Bemerkungen: Ist n so klein, dass keine Rekursion mehr stattfindet, so gilt die Formel natürlich nicht mehr, sondern $T(n) = O(1)$. Das Problem, dass n nicht durch b glatt teilbar ist, kann man ignorieren.

Ein Satz, mit dem sich Rekursionsgleichungen bequem lösen lassen.

1-22

▶ Satz: Master-Theorem

Die Funktion $T : \mathbb{R} \to \mathbb{R}$ genüge folgender Rekursion:

$$T(n) = \underbrace{a \cdot T(n/b)}_{\text{Herrschen-Aufwand}} + \underbrace{\Theta(n^t \log^d n)}_{\text{Teilen-Aufwand}}$$

mit $a \geq 1$, $b > 1$ und $t \geq 0$ sowie, falls $t = 0$, auch $d \geq 0$.
Wir nennen t den Teilungsexponenten und $h = \log_b a$ den Herrschaftsexponenten. Dann gilt:

1. *Teilungsexponent größer: Teilen-Aufwand überwiegt*
   Falls $h < t$, so gilt $T(n) = \Theta(n^t \log^d n)$.
2. *Exponenten gleich: Logarithmischer Anstieg*
   Falls $h = t$, so gilt $T(n) = \Theta(n^t \log^{d+1} n)$.
3. *Herrschaftsexponent größer: Herrschen-Aufwand überwiegt*
   Falls $h > t$, so gilt $T(n) = \Theta(n^h)$.

Beachte: $h \gtreqless t$ gilt genau dann, wenn $a/b^t \gtreqless 1$.

Beweis des Master-Theorems

1-23

Ein allgemeines Lemma

▶ Lemma

Sei $T(n) = a \cdot T(n/b) + \Theta(n^t \log^d n)$. Dann gilt

$$T(n) = n^h T(1) + \sum_{i=0}^{\log_b n} \Theta\Big(\big(\tfrac{a}{b^t}\big)^i n^t \log^d \tfrac{n}{b^i} \Big).$$

Beweis. Wir setzen einfach immer wieder die Rekursionsformel an:

$$\begin{aligned}
T(n) &= aT(n/b) + \Theta(n^t \log^d n) \\
&= a^2 T(n/b^2) + \Theta(an^t/b^t \log^d n/b) + \Theta(n^t \log^d n) \\
&= a^3 T(n/b^3) + \Theta(a^2 n^t/b^{2t} \log^d n/b^2) \\
&\qquad + \Theta(an^t/b^t \log^d n/b) + \Theta(n^t \log^d n) \\
&= \dots \\
&= \underbrace{a^{\log_b n}}_{=n^h} T(1) + \sum_{i=0}^{\log_b n} \Theta(a^i n^t/b^{ti} \log^d n/b^i).
\end{aligned}$$

□

1-24

Beweis des Master-Theorems
Der Fall eines großen Teilungsexponenten.

▶ **Lemma**
Falls $h < t$, so gilt $T(n) = \Theta(n^t \log^d n)$.

Beweis. Es galt $T(n) = n^h T(1) + \sum_{i=0}^{\log_b n} \Theta\Big(\big(\frac{a}{b^t}\big)^i n^t \log^d \frac{n}{b^i} \Big)$.

1. Wegen $h < t$ wird der Term $n^h T(1)$ von $n^t \log^d n$ für $i = 0$ »aufgefressen«.
2. Wegen $h < t$ gilt weiter $x := a/b^t < 1$ und somit

$$\sum_{i=0}^{\log_b n} \Theta\Big(\big(\frac{a}{b^t}\big)^i n^t \log^d \frac{n}{b^i} \Big) \le \sum_{i=0}^{\log_b n} x^i \Theta(n^t \log^d n)$$
$$= \Theta(n^t \log^d n) \underbrace{\textstyle\sum_{i=0}^{\log_b n} x^i}_{<\infty}.$$

□

1-25

Beweis des Master-Theorems
Der Fall gleicher Exponenten.

▶ **Lemma**
Falls $h = t$, so gilt $T(n) = \Theta(n^t \log^{d+1} n)$.

Beweis. Es galt $T(n) = n^h T(1) + \sum_{i=0}^{\log_b n} \Theta\Big(\big(\frac{a}{b^t}\big)^i n^t \log^d \frac{n}{b^i} \Big)$.
Der Term $n^h T(1)$ wird immer noch von $n^t \log^d n$ für $i = 0$ »aufgefressen«.
Wegen $h = t$ gilt $a/b^t = 1$ und somit lässt sich $T(n)$ schreiben als

$$\sum_{i=0}^{\log_b n} \Theta\big(n^t \log_b^d \tfrac{n}{b^i}\big) = \Theta(n^t) \sum_{i=0}^{\log_b n} \log_b^d \tfrac{n}{b^i}$$
$$= \Theta(n^t) \sum_{i=0}^{\log_b n} (\log_b n - i \log_b b)^d.$$

Johannes Faulhaber

Setzt man $x = \log_b n$, so ist die letzte Summe

- gerade die Summe der ersten x Zahlen (für $d = 1$),
- der ersten x Quadratzahlen (für $d = 2$),
- der ersten x Kubikzahlen (für $d = 3$) und so weiter.

Hierfür gilt nach der Formel von Faulhaber, dass diese Summe in $\Theta(x^{d+1})$ liegt. □

1-26

Beweis des Master-Theorems
Der Fall eines großen Herrschaftsexponenten.

▶ **Lemma**
Falls $h > t$, so gilt $T(n) = \Theta(n^h)$.

Beweis. Es galt $T(n) = n^h T(1) + \sum_{i=0}^{\log_b n} \Theta\Big(\big(\frac{a}{b^t}\big)^i n^t \log_b^d \frac{n}{b^i} \Big)$.
Mit $y = a/b^t$ lässt sich die Summe umschreiben zu

$$\Theta(n^t) \sum_{i=0}^{\log_b n} y^i (\log_b n - i)^d = \Theta(n^t y^{\log_b n}) \sum_{i=0}^{\log_b n} y^{i - \log_b n} (\log_b n - i)^d$$

Setzt man $k = \log_b n$, so kann man die Summe schreiben als $\sum_{i=0}^{k} i^d / y^i$, was wegen $y > 1$ konvergiert, also beschränkt ist. Nun gilt

$$n^t y^{\log_b n} = n^t a^{\log_b n} / b^{t \log_b n} = n^t n^{\log_b a} / n^t = n^h.$$

□

1.3.5 Die Klassifikation

Teilen-und-Herrschen + Master-Theorem = Klassifikation. 1-27

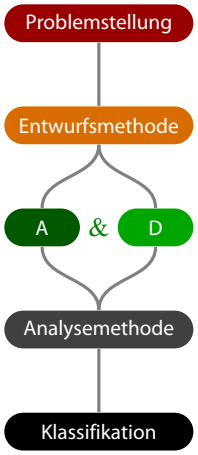
▶ Satz
Der Karatsuba-Algorithmus läuft in Zeit $\Theta(n^{\log_2 3})$.

Beweis. Die Laufzeit genügt der Rekursionsformel $T(n) = 3T(n/2) + O(n)$. Der Herrschaftsexponent ist somit $h = \log_2 3$ und der Teilungsexponent ist $t = 1$. Nach dem Fall »größerer Herrschaftsexponent« des Master-Theorems folgt die Behauptung. □

▶ Satz
Der Strassen-Algorithmus benötigt $\Theta(n^{\log_2 7})$ *Zahl-Multiplikationen zur Multiplikation zweier Matrizen.*

Beweis. Die Anzahl der Zahl-Multiplikationen genügt der Formel $T(n) = 7T(n/2) + O(n^2)$. Der Herrschaftsexponent ist somit $h = \log_2 7$ und der Teilungsexponent ist $t = 2$. Nach dem Fall »größerer Herrschaftsexponent« des Master-Theorems folgt die Behauptung. □

Zusammenfassung dieses Kapitels



Multiplikation von Zahlen
Multiplikation von Matrizen
Teile-und-Herrsche
Karatsuba-Algorithmus
Strassen-Algorithmus
Master-Theorem
Laufzeit $O(n^{\log_2 3})$
Laufzeit $O(n^{\log_2 7})$

1-28

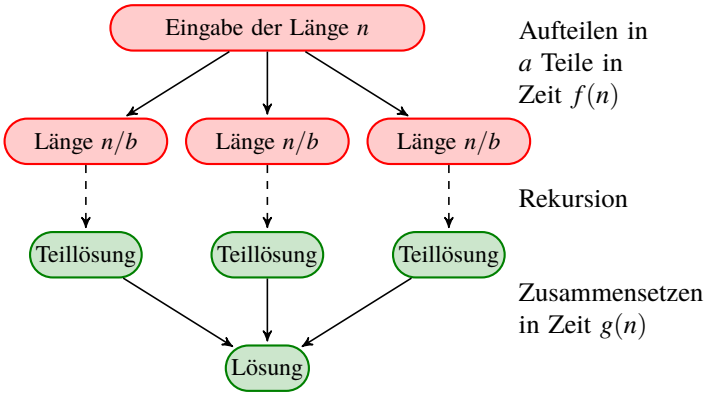
▶ Das Klassifikationswerkzeug »O-Klassen«
Sind f und g Funktionen, so entspricht *bis auf Faktoren und kleine Werte* grob:

$$\begin{aligned} f \in O(g) &\quad\approx\quad f \leq g, \\ f \in \Theta(g) &\quad\approx\quad f = g, \\ f \in \Omega(g) &\quad\approx\quad f \geq g. \end{aligned}$$

Die wichtigsten O-Klassen und ihre Inklusionsbeziehungen:

$$\begin{aligned} O(1) &\subsetneq O(\log n) \subsetneq O(\sqrt{n}) \\ &\subsetneq O(n) \subsetneq O(n \log n) \subsetneq O(n \log^2 n) \\ &\subsetneq O(n^2) \subsetneq O(n^3) \\ &\subsetneq O(2^n) \subsetneq O(3^n). \end{aligned}$$

▶ Das Entwurfsprinzip »Teilen-und-Herrschen«



Dies führt zur Rekursionsformel $T(n) = f(n) + aT(n/b) + g(n)$.

▶ Das Analysewerkzeug »Master-Theorem«

Sei

$$T(n) = \underbrace{a \cdot T(n/b)}_{\text{Herrschen-Aufwand}} + \underbrace{\Theta(n^t \log^d n)}_{\text{Teilen-Aufwand}}.$$

und $h = \log_b a$ der *Herrschaftsexponenten*. Dann gilt:

1. *Teilungsexponent größer: Teilen-Aufwand überwiegt*
   Falls $h < t$, so gilt $T(n) = \Theta(n^t \log^d n)$.
2. *Exponenten gleich: Logarithmischer Anstieg*
   Falls $h = t$, so gilt $T(n) = \Theta(n^t \log^{d+1} n)$.
3. *Herrschaftsexponent größer: Herrschen-Aufwand überwiegt*
   Falls $h > t$, so gilt $T(n) = \Theta(n^h)$.

▶ Die Algorithmen

1. Der Karatsuba-Algorithmus multipliziert Zahlen in Zeit $\Theta(n^{\log_2 3})$.
2. Der Strassen-Algorithmus multipliziert Matrizen in Zeit $\Theta(n^{\log_2 7})$.

Zum Weiterlesen

[1] Martin Fürer. Faster Integer Multiplication, *Proceedings of the 39th Annual Symposium on Theory of Computing,* ACM, 57–66, 2007.

Über das Problem, zwei Zahlen zu multiplizieren, kann man auch heute noch auf einer der wichtigsten Konferenzen der Informatik Artikel vorstellen (Theoretiker sind immer sehr glücklich, wenn sie bei STOC oder FOCS einen Artikel akzeptiert bekommen). In diesem Artikel stellt Fürer den derzeit schnellsten Algorithmus zur Multiplikation von Zahlen vor.

[2] Volker Strassen. Gaussian Elimination is not Optimal, *Numerische Mathematik,* 13: 354–356, 1969.

Dieser Artikel hat eine Länge von ganzen drei Seiten; in der Kürze liegt die Würze. Bis heute rätseln Forscher, wie Strassen auf die in dem Artikel angegebenen Gleichungen gekommen ist, sie sind dort – anders als in diesem Kapitel – ohne motivierendes Gedöns einfach angegeben.

Übungen zu diesem Kapitel

Übung 1.1 Schnelle Polynommultiplikation, schwer

Ein Polynom vom Grad n kann man durch $n+1$ Koeffizienten (Zahlen) beschreiben: Beispielsweise ist das Polynom $p(x) = x^2 - 2$ vom Grad 2 und wenn man es als $p(x) = 1 \cdot x^2 + 0 \cdot x^1 + (-2) \cdot x^0$ schreibt, so sieht man, dass das Polynom durch die Koeffizientenfolge $(1, 0, -2)$ eindeutig beschrieben ist.

Geben Sie einen Algorithmus mit an, der die Koeffizienten von zwei Polynomen p und q jeweils vom Grad n als Eingabe erhält und der die Koeffizienten ihres Produkts pq berechnet (das Produkt r von zwei Polynomen p und q ist auf die nahe liegende Art definiert durch die Vorschrift $r(x) = p(x) \cdot q(x)$). Ihr Algorithmus darf zur Berechnung des Produkts maximal $O(n^{\log_2 3})$ Multiplikationen von Zahlen durchführen.

Erläutern Sie Ihren Algorithmus kurz und führen Sie eine Analyse der Laufzeit durch.

Tipp: Schreiben Sie ein Polynom $p(x)$ als $p_1(x)x^{n/2} + p_2(x)$, wobei beide p_i den Grad $n/2$ haben.

Übung 1.2 Transitivität der O-Relation beweisen, leicht

Wenden Sie die Definition der O-Relation aus der Vorlesung an, um zu zeigen, dass für alle Funktionen $f, g, h \colon \mathbb{N} \to \mathbb{N}$ mit $f \in O(g)$ und $g \in O(h)$ auch $f \in O(h)$ gilt. Was folgt hieraus für die Ω- und Θ-Relationen?

Übung 1.3 Probleme und Algorithmen für typische Wachstumsklassen finden, leicht

Finden Sie zu jeder der folgenden typischen Wachstumsklassen ein Problem, welches sich durch einen Algorithmus lösen lässt, dessen Laufzeit oder Platzbedarf durch eine Funktion aus der Klasse beschränkt ist:

$$O(1) \subsetneq O(\log n) \subsetneq O(\sqrt{n}) \subsetneq O(n) \subsetneq O(n \log n) \subsetneq O(n^2) \subsetneq O(n^3) \subsetneq O(2^n) \subsetneq O(n!) \subsetneq O(n^n).$$

Die Probleme und Algorithmen sollten dabei leicht verständlich und kurz erklärbar oder aus dem Studium schon bekannt sein. Geben Sie für jedes Problem an, wofür n steht.

Übung 1.4 Algorithmus von Karatsuba anwenden, leicht

Wenden Sie den Algorithmus von Karatsuba an, um das Produkt $7421 \cdot 1211$ zu berechnen.

Übung 1.5 Master-Theorem zur Analyse von Algorithmen verwenden, leicht

In diesem Kapitel wurden die Laufzeiten der Algorithmen Merge-Sort, Quick-Sort und Binäre Suche durch Rekursionsgleichungen abgeschätzt.

Auf welche Gleichungen lässt sich das Master-Theorem anwenden? Ermitteln Sie für Algorithmen, auf die sich das Theorem anwenden lässt, eine asymptotische Abschätzung der Laufzeit.

Übung 1.6 Master-Theorem anwenden, leicht

Verwenden Sie das Master-Theorem, um das Wachstum der Funktion $T(n) = 2T(n/4) + g_i(n)$ für folgende $g_i \colon \mathbb{N} \to \mathbb{N}$ abzuschätzen: $g_1(n) = 1$, $g_2(n) = \log n$, $g_3(n) = \sqrt{n}$, $g_4(n) = \sqrt{n} \cdot \log n$, $g_5(n) = n$ und $g_6(n) = n^2$.

2-1

Kapitel 2
Entwurfsmethode: Dynamische Tabellen
Der Rekursionwolf im Iterationspelz

2-2

Lernziele dieses Kapitels

1. Beispiele von Problemstellungen kennen, die sich mit dynamischen Tabellen lösen lassen
2. Konzept der dynamischen Tabelle in eigenen Anwendungen einsetzen können
3. Auf dynamischen Tabellen aufbauende Algorithmen analysieren können

Inhalte dieses Kapitels

Worum es heute geht

Wäre Hamlet Informatik-Student gewesen und nicht ein wankelmütiger von Geistern geleiteter dänischer Prinz, so hätte er wohl seinen berühmtesten Monolog eingeleitet mit den Worten »To recurse or not to recurse, that is the question.« Wäre das Gretchen eine Informatik-Studentin gewesen, so hätte sie ihren an der Wissenschaft im Allgemeinen zweifelnden und sich deshalb der Esoterik hingebenden Lover Faust wohl gefragt »Nun sag, wie hast du's mit der Rekursion? Du bist ein herzlich guter Mann, allein ich glaub, du hältst nicht viel davon.«

Um kaum eine Frage wird und wurde in der Informatik so gerungen wie um die Frage: Lieber rekursiv oder iterativ? Die Beantwortung dieser Frage durch Informatikprofessorinnen und -professoren (die es ja eigentlich wissen sollten) ändert sich alle paar Jahre: Als die Informatik noch jung und knackig war, war Speicherplatz kostbar und Call-Stacks hatten Platz für vielleicht sechzehn Einträge. Folglich führten Rekursionen schnell zu »Stack-Overflows« und die Lehrbücher füllten sich mit Verfahren, wie man Rekursion durch Iterationen ersetzen kann. Als dann mit der funktionalen Programmierung die reine Lehre mehr en vogue kam und die Sprachen Call-Stacks mit Tausenden oder gar Millionen Einträgen zuließen, waren rekursive Ansätze plötzlich sehr schick. Man konnte in funktionalen Sprachen überhaupt nicht mehr iterative Programmieren – und die Lehrbücher füllten sich mit Verfahren, wie man For-Schleifen geschickt in Tail-Recursions umbasteln kann. Heute hat sich die Lage etwas entspannt: Moderne Sprachen bieten einerseits sowohl Rekursion wie Iteration als gut nutzbare syntaktische Konstrukte an und die Call-Stacks sind so groß, dass auch Rekursionen mit einigen Hundert Rekursionsstufen verkraftet werden.

Ist es also egal, ob man nun rekursiv oder iterativ vorgeht? Sollte man immer dem eleganteren Konstrukt den Vortritt lassen?

Nein, sollte man nicht.

Es gibt Fälle, bei denen rekursive Ansätze elegant sind, aber kläglich scheitern. Der prominenteste Fall ist die Fibonacci-Folge:

$$f(n) = \begin{cases} 1 & \text{für } n = 1 \text{ und für } n = 2 \text{ und} \\ f(n-1) + f(n-2) & \text{für } n \geq 3. \end{cases}$$


GNU Free Documentation Licence

Eleganter kann man es kaum noch ausdrücken. Wandelt man dies aber direkt in Java-, C-, Scala-, Lua- oder Was-auch-immer-Code um, so erhält man ein irrsinnig langsames Programm. Der triviale iterative Ansatz (in einer Schleife den jeweils nächsten Wert aufgrund der beiden vorherigen ausrechnen) führt hingegen zu einem schnellen Programm (es geht allerdings auch noch viel schneller).

Woran liegt dies? Was macht den iterativen Ansatz schnell? Ist es der vermiedene Aufwand durch Funktionsaufrufe? Liegt es an weniger Caching-Misses, da weniger Speicher benötigt wird? Der Grund liegt tiefer: Der iterative Ansatz *vermeidet doppelte Berechnungen derselben Werte*. Das rekursive Programm »merkt« nicht, wenn es beispielsweise zur Berechnung von $f(7)$ den Wert $f(6)$ ausrechnet, dass es dies gerade schon einmal getan hat, als es nämlich für die Berechnung von $f(8)$ auch schon $f(6)$ ausgerechnet hat.

Man könnte die rekursive Berechnung der Fibonacci-Folge unglaublich beschleunigen, vermiede man dieses Neuberechnen schon einmal berechneter Werte. Genau an dieser Stelle setzen die dynamischen Tabellen an, um die es heute gehen soll: Eine dynamische Tabelle ist im Prinzip nichts anderes als eine Tabelle, in der Werte so gespeichert sind, dass man bei rekursiven Aufrufen nichts doppelt berechnet. Für die Fibonacci-Folge bedeutet dies konkret Folgendes: Sobald wir in der Rekursion einen Wert für ein Folgenglied errechnet haben, speichern wir dies in einem Array. Immer wenn nun ein rekursiver Aufruf erfolgen soll, so schauen wir zunächst nach, ob wir diesen Wert nicht schon in der Tabelle stehen haben. Wenn ja, so rechnen wir einfach mit diesem weiter und sparen uns die Rekursion.


Gemeinfrei

Man sieht leicht, dass man im Resultat einen Algorithmus erhält, der genauso schnell ist wie der iterative Algorithmus. Eine dynamische Tabelle ist letztendlich ein Rekursionswolf im Iterationspelz. Es sei erwähnt, dass man sich in der besten aller Welten gar nicht um die Erstellung der dynamischen Tabellen verdient machen braucht: Soll sich doch der Compiler darum kümmern! Wenn man nämlich sein Programm in einer reinen funktionalen Sprache schreibt, so kann sich der Compiler, wenn er denn will, die Ergebnisse einer Funktionsauswertung einfach selber in einer Tabelle merken (man spricht von *Memoization*). In diesem Fall beschreibt man das Problem so wie oben rekursiv in vollendeter Eleganz und erhält ein Programm, das so schnell ist wie der iterative Ansatz.

Womit wir wieder beim Ausgangspunkt wären und die Schlacht zwischen Rekursion und Iteration in eine neue Runde geht.

2.1 Der Weg zur »dynamischen Tabelle«

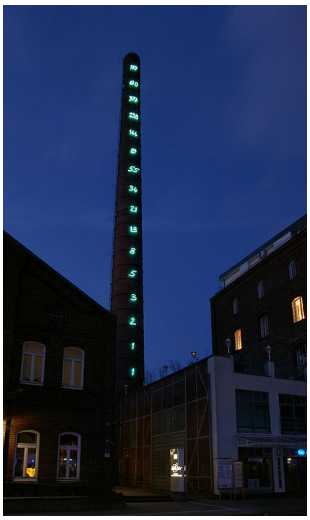
2.1.1 Von der Rekursion. . .

Der Klassiker unter den Folgen.

2-4

Die *Fibonacci-Folge* ist rekursiv wie folgt definiert:

$$f(n) = \begin{cases} 1 & \text{für } n \leq 2, \\ f(n-1) + f(n-2) & \text{für } n \geq 3. \end{cases}$$


Author Rainer Knäpper, Creative Commons Attribution Sharealike Licence

Dies lässt sich leicht in Java implementieren:

```
int fib(int n)
{
  if (n<=2)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

✎ Zur Diskussion
Wie verhält sich diese Laufzeit im Vergleich zu einer iterativen Lösung?

2-5

Ein zweites Beispiel: Lösen von Rekursionsgleichungen.

Wir wollen eine Rekursionsgleichung lösen wie die Folgende:

$$T(1) = 23,$$
$$T(2) = 42,$$
$$T(n) = T(\lceil n/3 \rceil) + 8T(\lfloor n/3 \rfloor) + 12n^2 + 3.$$

Mit dem *Master-Theorem* lässt sich diese Rekursion *leicht lösen*:

1. Wir schreiben dies zunächst um als $T(n) = 9T(n/3) + \Theta(n^2)$.
2. Der Herrschaftsexponent ist $h = \log_3 9 = 2$.
3. Der Teilungsexponent ist $t = 2$.
4. Also liegt die Lösung in $\Theta(n^2 \log n)$.

Will man *aber nicht nur die Theta-Klasse* bestimmen, sondern *die Funktion genau ausrechnen*, so kann man dies leicht rekursiv machen.

```
int T(int n)
{
  if (n<=1) return 23;
  if (n==2) return 42;
  return T(n%3 == 0 ? n/3:n/3+1) + 8*T(n/3) + 12*n*n + 3;
}
```

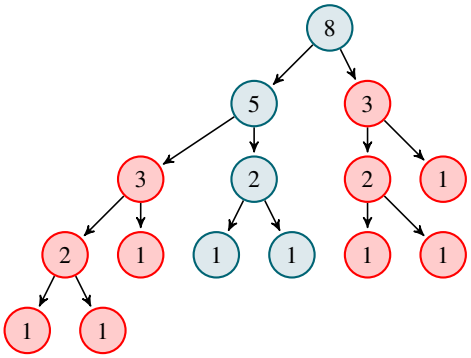
✎ Zur Diskussion
Wie verhält sich diese Laufzeit im Vergleich zu einer iterativen Lösung?

2.1.2 . . . über Memoization. . .

2-6

Warum ist die rekursive Lösung so langsam?

Das zentrale Problem der Rekursion ist, dass *die gleichen Werte immer wieder berechnet werden*.



2-7

Dies kann man vermeiden durch »*Caching*«.

Big Idea

Immer, wenn man einen Funktionswert berechnet hat, speichert man diesen in einer *Tabelle*. Wenn der Funktionswert erneut benötigt wird, so *spart man sich die Berechnung* und gibt den Wert in der Tabelle zurück.

```
boolean[] cached;
int[]     table;

int fib(int n)
{
  if (!cached[n]) {
    if (n<=2) table[n] = 1;
    else      table[n] = fib(n-1) + fib(n-2);
```

```
    cached[n] = true;
  }
  return table[n];
}
```

Diesen Trick nennt man auch *Memoization* (»Erinnerung«).

```
boolean[] cached;
int[]     table;

int T(int n)
{
  if (!cached[n]) {
    if (n<=1)      table[n] = 23;
    else if (n==2) table[n] = 42;
    else           table[n] =
        T(n%3 == 0 ? n/3:n/3+1) + 8*T(n/3) + 12*n*n + 3;
    cached[n] = true;
  }
  return table[n];
}
```

2.1.3 . . . zur Iteration

Bottom-Up versus Top-Down. 2-8

Eine Beobachtung zur Berechnung

Betrachtet man Memoization genauer, so geht die Rekursion zwar »top-down« vor, die Tabelle füllt sich aber »bottom-up« (da die kleinsten Werte als erstes vorliegen).

Dynamische Tabellen: Iterationen, die mal Rekursionen waren

Wenn sich die Tabelle »sowieso« von unten nach oben füllt, dann ist es *oft schneller*, diese *in einer Schleife* von unten nach oben zu füllen. Das resultierende *iterative Programm* nennt man (komischerweise) ein *»dynamisches Programm«* und die Tabelle eine *»dynamische Tabelle«*.

Die Fibonacci-Folge mittels einer dynamischen Tabelle. 2-9

```
int fib(int n)
{
  int[] table = new int[n+1];

  for (int i = 1; i <= n; i++) {
    if (i<=2) table[i] = 1;
    else      table[i] = table[i-1] + table[i-2];
  }
  return table[n];
}
```

(Der Profi »sieht« natürlich, dass man gar nicht die komplette Tabelle speichern muss, da immer nur die letzten beiden Element benötigt werden.)

2-10

✎ Zur Übung

Schreiben Sie den Java-Code eines dynamischen Programms auf, das die Funktion T berechnet:

```
int T(int n)
{
  int[] table = new int[n+1];

  for (int i = 1; i <= n; i++) {
    // Ihr Code




  }
  return table[n];
}
```

2.2 Dynamische Tabellen für Zahl-Probleme

2.2.1 Das Subset-Sum-Problem

2-11

Zwei »einfache« Probleme

Auf einer Tafel mögen ein paar Zahlen stehen:

3170	9794	932	3889	8001	2113	987	8867	345
9347	2690	7669	1092	4790	3049	433	7235	
8647	8663	986	4116	9044	7031	3538	1218	
4935	5953	6131	1791	5720	8828	1882	7229	
1242	4424	6540	3541	8915	7081	2211	4261	
8084	3635	6752	4158	5002	3491	3277	4662	
5578	6689	3708	3722	1896	5900	3814	9271	
3069	23	1455	3231	6416	5014	2175	1489	364
1616	1982	9954	122					

Zwei Aufgaben

1. Unterstreichen Sie möglichst wenige Zahlen, so dass deren Summe *mindestens* 100.000 ergibt.
2. Unterstreichen Sie möglichst wenige Zahlen, so dass deren Summe *genau* 100.000 ergibt.

2-12

Das formale Subset-Sum-Problem

Die formale Problemstellung

Eingabe Eine Folge $(a_1, \ldots, a_n)$ von natürlichen Zahlen und eine Zahl b.

Frage Gibt es eine Teilmenge $I \subseteq \{1, \ldots, n\}$, so dass $\sum_{i \in I} a_i = b$.

Die ganz formale Problemstellung

$$\textsc{subsetsum} = \big\{ \mathrm{bin}(a_1)\#\ldots\#\mathrm{bin}(a_n)\#\#\mathrm{bin}(b) \;\big|\; \exists I \subseteq \{1,\ldots,n\} \big(\textstyle\sum_{i \in I} a_i = b \big) \big\}.$$

Anwendung der Entwurfsmethode »dynamische Tabelle« auf das Subset-Sum-Problem. 2-13

Schritt 1: Das rekursive Programm

Das Problem lässt sich rekursiv sehr leicht lösen:

```
1 function subsetsum(a_1,...,a_n,b) : boolean
2   if n = 0 then
3     return b = 0
4   else
5     return subsetsum(a_1,...,a_{n-1},b) ∨ subsetsum(a_1,...,a_{n-1},b-a_n)
```

In Java:

```
boolean subsetsum(int[] a, int n, int b)
{
  if (b < 0) return false;

  if (n == 0)
    return b==0;
  else
    return subsetsum(a,n-1,b) || subsetsum(a,n-1,b-a[n]);
}
```

Anwendung der Entwurfsmethode »dynamische Tabelle« auf das Subset-Sum-Problem. 2-14

Schritt 2: Memoization

```
boolean[][] cached;
boolean[][] table;

boolean subsetsum(int[] a, int n, int b)
{
  if (b < 0) return false;

  if (!cached[b][n]) {
    if (n == 0)
      table[b][n] = b==0;
    else
      table[b][n] = subsetsum(a,n-1,b)
                  || subsetsum(a,n-1,b-a[n]);
    cached[b][n] = true;
  }
  return table[b][n];
}
```

Anwendung der Entwurfsmethode »dynamische Tabelle« auf das Subset-Sum-Problem. 2-15

Schritt 3: Umschreiben als Iteration

```
boolean subsetsum(int[] a, int n, int b)
{
  boolean[][] table = new boolean[b+1][n+1];

  for (int k = 0; k <= n; k++) {
    for (int i = 0; i <= b; i++) {
      if (k == 0)
        table[i][k] = i==0;
      else
        table[i][k] = table[i][k-1] ||
                      (i>=a[k] ? table[i-a[k]][k-1] : false);
    }
  }
  return table[b][n];
}
```

Wieder kann man Platz sparen, wenn man beachtet, dass immer nur auf Tabelleneinträgen für $k-1$ zugegriffen wird. Es reicht also ein eindimensionaler Array.

2-16

Die Laufzeit des Dynamische-Tabellen-Algorithmus für das Subset-Sum-Problem.

▶ Satz
Der auf einer dynamischen Tabelle basierende Algorithmus löst das Subset-Sum-Problem in Zeit $O(n \cdot b)$.

Beweis. Der Algorithmus iteriert über zwei verschachtelte Schleifen, mit $n+1$ beziehungsweise $b+1$ Iterationen. □

Bemerkungen:

- Dies bedeutet *nicht*, dass SUBSETSUM $\in$ P. Warum nicht?
- Algorithmen mit Laufzeiten, die polynomiell in den Werten der vorkommenden Zahlen sind (nicht aber unbedingt in der Länge der Kodierung der Eingabe), nennt man *pseudopolynomiell*.

2.2.2 Das Rucksack-Problem

2-17

Das Dilemma des Diebes.

Der Dieb und sein Rucksack.

Ein Dieb knackt einen Tresor, in dem sich Gold, Juwelen und Edelsteine befinden. In seinem Rucksack kann er aber nur maximal 50kg tragen.
Welche Gegenstände sollte er einpacken, um deren Gesamtwert zu maximieren?

Die formale Problemstellung

Eingabe Eine Folge $(w_1,\ldots,w_n)$ von *Werten* und eine Folge $(g_1,\ldots,g_n)$ von *Gewichten* und ein Maximalgewicht m.

Ausgabe Der maximale Wert von $\sum_{i\in I} w_i$ über alle Teilmengen $I \subseteq \{1,\ldots,n\}$, so dass $\sum_{i\in I} g_i \leq m$.

2-18

Anwendung der Entwurfsmethode »dynamische Tabelle« auf das Rucksack-Problem.

Schritt 1: Das rekursive Programm

```
1 function rucksack(w_1,...,w_n,g_1,...,g_n,m) : int
2   if n = 0 then
3     return 0
4   else
5     if m < g_n then // Passt nicht?
6       return rucksack(w_1,...,w_{n-1},g_1,...,g_{n-1},m)
7     else // Passt
8       return max{rucksack(w_1,...,w_{n-1},g_1,...,g_{n-1},m),
9                  w_n + rucksack(w_1,...,w_{n-1},g_1,...,g_{n-1},m-g_n)}
```

In Java:

```
int rucksack(int[] w, int[] g, int n, int m)
{
  if (n == 0)
    return 0;
  else if (m < g[n])
    return rucksack(w,g,n-1,m);
  else
    return Math.max(rucksack(w,g,n-1,m),
                    w[n]+rucksack(w,g,n-1,m-g[n]));
}
```

Anwendung der Entwurfsmethode »dynamische Tabelle« auf das Rucksack-Problem.

2-19

Schritt 2: Memoization

```
boolean[][] cached;
int[][]     table;

int rucksack(int[] w, int[] g, int n, int m)
{
  if (!cached[m][n]) {
    if (n == 0)
      table[m][n] = 0;
    else if (m < g[n])
      table[m][n] = rucksack(w,g,n-1,m);
    else
      table[m][n] = Math.max(rucksack(w,g,n-1,m),
                             w[n]+rucksack(w,g,n-1,m-g[n]));
    cached[m][n] = true;
  }
  return table[m][n];
}
```

Anwendung der Entwurfsmethode »dynamische Tabelle« auf das Rucksack-Problem.

2-20

Schritt 3: Als Iteration

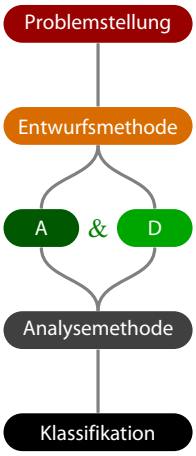
```
int rucksack(int[] w, int[] g, int n, int m)
{
  int[][] table = new int[m+1][n+1];

  for (int k=0; k<=n; k++) {
    for (int i=0; i<=m; i++) {
      if (k == 0)
        table[i][k] = 0;
      else if (i < g[k])
        table[i][k] = table[i][k-1];
      else
        table[i][k] = Math.max(table[i][k-1],
                    w[k] + table[i-g[k]][k-1]);
    }
  }
  return table[m][n];
}
```

▶ Satz

Das Rucksack-Problem kann in Zeit $O(mn)$ gelöst werden.

Zusammenfassung dieses Kapitels

2-21



Subset-Sum-Problem
Knapsack

Dynamische-Tabellen-Methode / Memoization

Dynamische Tabelle

Einfache Schleifenanalyse

Pseudopolynomielle Laufzeit

▶ Die Entwurfsmethode: Dynamische Tabellen

Dynamische Tabellen entstehen, wenn man

1. auf ein rekursives Programm
2. die Technik der Memoization anwendet und
3. daraus wiederum ein iteratives Programm macht.

Wer darin geübt ist, kann natürlich auch gleich das iterative Programm hinschreiben.

Merke

Dynamische Tabellen bringen nur dann etwas, wenn ein rekursives Programm die immer gleichen Werte in verschiedenen Rekursionsästen berechnen würde.

▶ Die Klassifikation

Es gelten die folgenden Sätze:

▶ Satz

Das Subset-Sum-Problem kann mittels dynamischer Tabellen in Zeit $O(n \cdot b)$ und Platz $O(b)$ gelöst werden.

▶ Satz

Das Rucksack-Problem kann mittels dynamischer Tabellen in Zeit $O(n \cdot m)$ und Platz $O(m)$ gelöst werden.

Achtung

Man kann zeigen, dass beide Probleme NP-vollständig sind!

Zum Weiterlesen

[1] Daniel Lokshtanov, Jesper Nederlof, Saving space by algebraization. *Proceedings of the 42nd ACM Symposium on Theory of Computing* (STOC), 321–330, 2010.

Wir haben in diesem Kapitel gesehen, dass man das Subset-Sum-Problem in pseudopolynomieller Zeit und pseudopolynomiellem Platz lösen kann. Es ist auch leicht zu sehen, dass man das Problem in polynomiellem Platz lösen kann. In diesem Paper wird nun gezeigt, dass man durch »Algebraisierung« einen Algorithmus erhält, der das Problem gleichzeitig in pseudopolynomieller Zeit und polynomiellem Platz löst.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms,* zweite Auflage, MIT Press, 2001, Kapitel 15 »Dynamic Programming«.

In diesem Kapitel wird das Konzept der dynamischen Tabellen sehr schön erklärt mit vielen interessanten Beispielen.

Übungen zu diesem Kapitel

Übung 2.1 Planende Diebe, mittel, mit Lösung

Meisterhacker A. Nonymous plant seinen nächsten großen Coup: Er wird mal wieder jede Menge Daten der Firma S. Ony klauen und diese dann verkaufen. Er hat es bereits geschafft, sich vom Server der Firma S. Ony ein Verzeichnis der Daten zu beschaffen, die er stehlen könnte: Er kennt für jede Datei deren Größe in Byte und aufgrund ihrer Namen weiß er auch, wie viel Euro jede Datei wert sein wird.

A. Nonymous hat vor, Dateien im Wert von genau einer Million Euro zu klauen. Da der Diebstahl um so schwieriger wird, je mehr Daten er vom Server überträgt, fragt er sich, wie viele Byte er mindestens übertragen muss, um Daten im Wert von genau einer Million Euro zu bekommen.

Helfen Sie A. Nonymous! Geben Sie dazu zunächst einen rekursiven Algorithmus an, der zwei Arrays g und w jeweils der Länge n als Eingabe bekommt, wobei g die Größen der Dateien enthält und w ihre Werte, sowie eine Zahl r (im konkreten Fall eine Million). Ergebnis soll der kleinste Wert $\sum_{i\in I} g[i]$ sein über alle $I \subseteq \{1,\ldots,n\}$ so dass $\sum_{i\in I} w[i] = r$.

Übung 2.2 Alternatives dynamisches Programm für das Rucksackproblem, mittel

Geben Sie ein Programm an, das das Rucksackproblem in Zeit $O(nW)$ löst, wobei W die Summe aller Werte von Gegenständen im Tresor ist, also $W = \sum_{i=1}^{n} w_i$.

Tipp: Wandeln Sie den Algorithmus aus Übung 2.1 zunächst in ein dynamisches Programm um. Ermitteln Sie dann das größte r, so dass im Resultatarray an der Stelle r der Eintrag die Rucksackgröße m nicht überschreitet.

Übung 2.3 Unabhängige Mengen berechnen, mittel

Für einen Graphen $G = (V,E)$ ist eine *unabhängige Knotenmenge* eine Teilmenge der Knoten $U \subseteq V$, zwischen denen es keine Kanten in G gibt (das heißt, für alle Paare von Knoten $v,w \in U$ gilt $\{v,w\} \notin E$.) Für die in dieser Aufgabe betrachteten Graphen hat jeder Knoten $v \in V$ zudem einen Wert $w(v) \in \mathbb{N}$. Im Folgenden soll die Entwurfsmethode der dynamischen Tabellen verwendet werden, um möglichst wertvolle unabhängige Mengen zu finden. Das heißt, für einen gegebenen Graphen $G = (V,E)$ ist eine unabhängige Menge $U \subseteq V$ gesucht, die die Summe $\sum_{v\in U} w(v)$ maximiert.

1. Entwickeln Sie einen Algorithmus, der möglichst wertvolle unabhängige Mengen in Pfad-Graphen findet.
2. Erweitern Sie ihn, um möglichst wertvolle unabhängige Mengen in Bäumen zu finden.

Übung 2.4 Dynamische Programmieransätze für das Rucksackproblem implementieren, mittel

Das Ziel dieser Aufgabe ist es, ein Programm zu schreiben, welches das Rucksackproblem löst. Ihr Programm sollte dabei so effizient sein, dass es die Testeingaben von der Webseite der Veranstaltung in kurzer Zeit lösen kann. Um alle Testeingaben lösen zu können, empfiehlt es sich, auf verschiedene Lösungsansätze für das Rucksackproblem zurückzugreifen und für eine gegebene Eingabe jeweils die passendste auszuwählen: An Lösungsansätzen kennen sie bereits

1. die dynamische Tabelle über das Gewicht des Rucksackinhaltes aus der Vorlesung,
2. die dynamische Tabelle über den Wert des Rucksackinhaltes aus Aufgabe 2.2 und
3. die rekursive Methode zum Lösen des Rucksackproblems, die alle möglichen Rucksackinhalte durchgeht.

Kombinieren Sie diese und gegebenenfalls weitere Ansätze, um die Testeingaben möglichst schnell zu lösen.

Jede Eingabedatei hat die folgende Form, wobei in der ersten Zeile die Größe m des Rucksacks und in den weiteren Zeilen jeweils zuerst das Gewicht g_i und dann der Wert w_i eines Gegenstandes steht.

```
3
1 1
2 2
2 3
1 2
```

Ihr Programm soll für eine solche Datei den größtmöglichen Wert eines Rucksackinhaltes berechnen. Für die Instanz aus der obigen Datei ist dieser Wert zum Beispiel 5.

Hinweis: Sie dürfen davon ausgehen, dass alle Zahlen in ein Java `long` passen.

Teil II

Offline-Probleme 2: Texte

Wie weit waren Sie in Ihrem Leben jemals von mit Text bedruckten oder beschriebenen Flächen entfernt?

Vermutlich werden Sie es kaum auf zehn Meter geschafft haben. Es fängt schon damit an, dass Ihre Kleidung voll ist mit kleinen Hinweisschildern und Texten; selbst eine Badehose hat noch eine Kochanleitung eingedruckt. Praktisch die einzige Chance, einen größeren Abstand von Texten zu gewinnen, ist, nackt in einem Gebirgssee zu schwimmen. Ohne Boote in der Nähe.

Nichts durchdringt unsere Zivilisation so sehr wie Symbolfolgen; sie sind im wahrsten Sinne des Wortes allgegenwärtig.

Es ist daher auch kein Wunder, dass in der Informatik Texte beziehungsweise »Strings« zu den wichtigsten Konzepten überhaupt gehören. Auf der praktischen Seite ist es in so ziemlich jeder Programmiersprache möglich, Strings direkt einzugeben, sie zu manipulieren und effizient zu verarbeiten. Auf der theoretischen Seite sind Zeichenketten *das* mathematische Grundkonstrukt, auf dem beispielsweise die gesamte Komplexitätstheorie aufgebaut ist.

Man könnte im Rahmen des Algorithmendesigns den Zeichenketten ganze Vorlesungsreihen widmen; da wir aber auch noch andere spannende Gebiete betrachten wollen, will ich nur zwei Themen herauspicken und anreißen: Das Indizieren von Texten und das Komprimieren von Texten.

3-1

Kapitel 3
A&D: Indizieren mit Bäumen
Von der Kunst, einen Index zu erstellen

Lernziele dieses Kapitels

1. Die Datenstruktur des Tries kennen und implementieren können
2. Die Datenstruktur des Suffix-Trees kennen
3. Methoden zur Suche in Texten kennen, die auf diesen Datenstrukturen aufbauen

Inhalte dieses Kapitels

3-2

Worum es heute geht

Die Suche in Texten gehört zum Brot-und-Butter-Geschäft moderner Rechner und es ist nicht immer der Mensch (neudeutsch User), der sich nicht mehr erinnern kann, von wem die Mail zum Thema »Extrem wichtige Besprechung« kam. Ständig wollen auch Programme in langen Texten irgendetwas suchen, seien es Schlüsselwörter, XML-Tags oder Variablennamen.

Wenn man lediglich einmal in einem Text etwas suchen möchte, so gibt es eine Reihe von mehr (Boyer-Moore) oder weniger (naive) raffinierten Suchalgorithmen. Oft kommt es aber auch vor, dass man im selben Text viele Male suchen möchte; das gilt für Wörterbücher ebenso wie für die Basensequenz von Viren. In diesem Fall liegt es nahe, einen speziellen *Index* zu erstellen, der die Suche etwas vereinfacht.

Die Idee, einen Index zu erstellen, um Suchanfragen zu beschleunigen, ist nicht neu. Ein Telefonbuch ist beispielsweise ein Index, auch wenn Telefonbücher akut vom Aussterben bedroht sind. Der Grund dafür ist aber nur, dass Computer eben die Indizierung für uns übernehmen.

In diesem Kapitel geht es um die Frage, wie sich schnell ein Index für einen langen Text erstellen lässt. Die im Laufe des Kapitels entwickelte Datenstruktur der »Suffix-Trees« kann sich sehen lassen: Für einen *beliebig* langen Text kann man einen Index erstellen, der

1. genauso groß ist wie der Text selber und
2. in dem man jedes beliebige Suchwort in einer Laufzeit findet, die linear von der Länge der zu suchenden Wortes abhängt und *völlig unabhängig* ist von der Länge des Textes.

Sucht man also mittels eines Suffix-Tree im menschlichen Genom (3 Milliarden Basepaare) nach einer Basesequenz der Länge 50, so benötigt dies nur 50 Zeitschritte.

3.1 Einführung

3.1.1 Fallbeispiel I: Bibliometrie

3-4

Welche Autoren schreiben über dasselbe Thema?

In der *Bibliometrie* geht es darum, *Maße* für Veröffentlichungen zu bestimmen. Beispielsweise kann man versuchen zu messen, wie »ähnlich« die Veröffentlichungen von zwei Autoren sind. Hierdurch kann man »Cluster« von Autoren bestimmen, aber auch Plagiate finden.

Wie bestimmt man schnell die Ähnlichkeit von Texten?

- Ein Maß der Ähnlichkeit von zwei Texten könnte sein, zu zählen, *wie häufig Worte in beiden Texten vorkommen*.
- Hat man zwei Texte mit je hundert Seiten, so sollen wir *schnell* für jedes Wort des einen Textes herausfinden, ob es in dem anderen Text vorkommt.

3.1.2 Fallbeispiel II: Virusdatenbanken

3-5

Die RNA-Virus-Datenbank

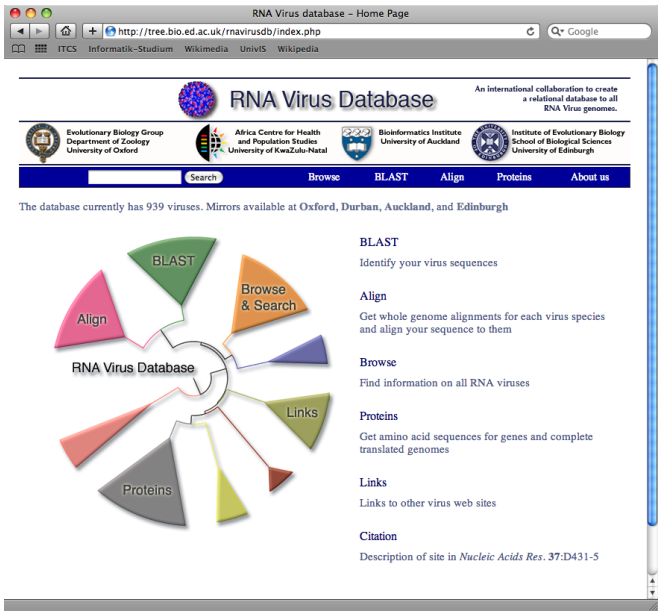
Eine RNA-*Virus-Datenbank* speichert die Genome von *Viren*:

Anfragen an eine RNA-Virus-Datenbank

- Das Blut einer Person oder eines Organismus wird untersucht.
- Man isoliert RNA-*Schnipsel* im Blut.
- Ziel ist es nun herauszufinden, *ob und wenn von welchem* Virus die Schnipsel stammen.

Wie findet man schnell Schnipsel?

- Die RNA-Virus-Datenbank enthält aktuell 939 Viren.
- Ein Viren-Genom ist zwischen 3.000 und ca 500.000 Basen lang.
- Wir sollen also *schnell und fehlertolerant* einen Schnipsel unter *hunderten Megabasenpaaren* finden.



Screenshot by Till Tantau

3.2 Tries

Die besonderen Eigenschaften von Strings. 3-6

Zur Verwaltung von *Strings* eigenen sich Suchbäume und Hash-Tabellen nur bedingt:

Probleme bei Hash-Tabellen zur String-Verwaltung

- Es dauert lange, einen Hash-Wert eines längeren Strings auszurechnen.
- Ein »fehlertolerantes« Suchen ist nicht möglich.

Probleme bei binären Suchbäumen zur String-Verwaltung

- Der lexikographische Vergleich ähnlicher Strings ist langsam.
- Fehlertolerantes Suchen ist schwierig.

3.2.1 Die Idee

Die Idee hinter Tries. 3-7

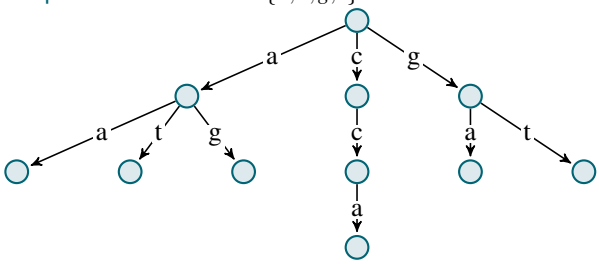
Ein Trie (von »information re*trie*val«) ist eine Datenstruktur *speziell zur Verwaltung vieler Strings*.

▶ **Definition:** Trie

Sie Σ ein Alphabet. Ein *Trie über* Σ ist ein gerichteter Baum mit folgenden Eigenschaften:

1. Jede Kante ist mit einem Symbol aus Σ beschriftet.
2. Für jeden Knoten gibt es für jedes Symbol maximal eine ausgehende Kante, die so beschriftet ist.

Beispiel: Ein Trie über $\Sigma = \{a, c, g, t\}$



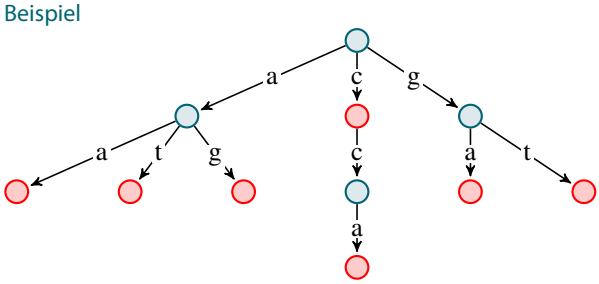
3.2.2 Grundoperationen

Ein Trie kann eine Menge von Strings speichern. 3-8

Man kann Tries benutzen, um eine *Menge von Strings* zu speichern:

- Einige Knoten des Tries werden *markiert*.
- Jeder Pfad von der Wurzel zu einem markierten Knoten *entspricht einem String*.

Beispiel



Dieser Trie speichert die Menge $\{aa, at, ag, c, cca, ga, gt\}$.

3-9

Die Grundoperationen auf Tries

Suchen eines Strings in einem Trie

```
function search (String w, Trie T) : boolean
  c ← root(T)
  for i ← 1 to length(w) do
    if c = null then
      return false
    else
      c ← child of c with edge label w[i]
  return true if c is marked, otherwise false
```

Einfügen eines Strings in einen Trie

```
algorithm insert (String w, Trie T)
  c ← root(T)
  for i ← 1 to length(w) do
    if c has no child with edge label w[i] then
      create new child of c with edge label w[i]
    c ← child of c with label w[i]
  mark c
```

```
algorithm delete (String w, Trie T)
  // Search
  c ← root(T)
  for i ← 1 to length(w) do
    if c = null then
      return
    else
      c ← child of c with edge label w[i]
  if c is not marked then
    return
  else
    // Ok, we found w
    unmark c
    // If c was a leaf, remove it and its parents up to the next unmarked node
    while c is a leaf and c is not marked do
      p ← parent of c
      delete c
      c ← p
```

3-10

✎ Zur Übung

1. Erstellen Sie einen Trie für folgende Menge von Strings: $\{barfoo, foobar, foo, bar, barfuss, fool\}$.
2. Wie lange dauert eine Suche nach einem String w der Länge $|w|$ in diesem Trie
   2.1 *höchstens* und
   2.2 *mindestens*?

   (Die »Laufzeit« sei gemessen in der Anzahl der untersuchten Knoten.)

3-11

Die Grundoperationen auf Tries.

▶ Satz

Einen String w in einen Trie der Größe gt

- *einzufügen dauert $O(|w|)$,*
- *zu löschen dauert $O(|w|)$ und*
- *zu suchen dauert $O(|w|)$.*

▶ Satz
Ein Trie für eine Menge M von Strings hat eine maximale Größe von $\sum_{w\in M}|w|$.

Bemerkungen

- Die oberen Schranken für die Laufzeiten sind *unabhängig von der Größe des Tries gt*.
- Auch Hashing kann keine besseren Laufzeiten liefern, da es $\Theta(|w|)$ dauert, den Hash-Wert eines Strings w zu berechnen.
- Man benötigt schon $\sum_{w\in M}|w|$ Platz, um die Elemente der Menge M überhaupt im Speicher zu halten.

Tries im Einsatz: Textvergleich

3-12

Wir können Tries wie folgt nutzen, um herauszufinden, *wie viele Worte zwei Texte gemeinsam haben*:

- Durchlaufe den ersten Text. Füge dabei jedes gefundene Wort in einen Trie ein.
- Durchlaufe nun den zweiten Text und überprüfe für jedes gefundene Wort, ob es im ersten Text vorkommt.

Offenbar benötigt dieses Verfahren Zeit $O(n+m)$, wenn die Texte die Längen n und m haben.

3.2.3 Pfad-Kompression: Patricia-Bäume

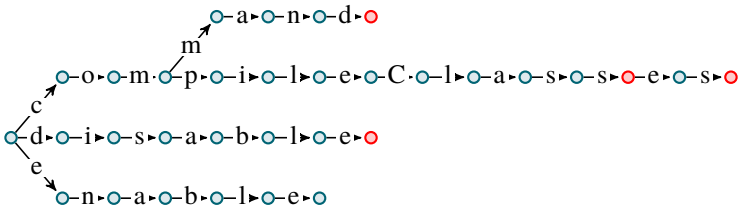
Ein Trie für Methodennamen

3-13

Nehmen wir an, wir bauen einen Trie für die *Methodennamen in einem Java-Programm*. Dann hätte dieser folgende Form:

- Nahe der Wurzel verzweigt er stark, da die Namen ganz unterschiedlich anfangen.
- Später gibt es dann aber immer wieder *lange Ketten* von Knoten.

Beispiel: Trie für die Methoden der Java-Klasse `java.lang.Compiler`



Patricia-Bäume sind komprimierte Tries.

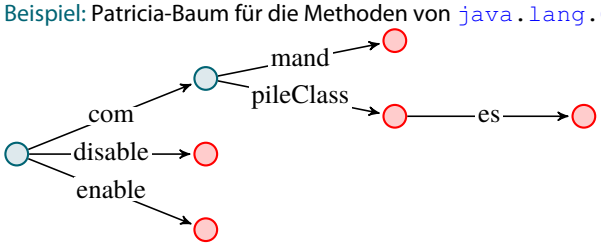
3-14

▶ Definition: Patricia-Baum
Ein *Patricia-Baum* ist ein Baum, der aus einem Trie entsteht, indem man

1. alle Pfade im Trie ohne Verzweigungen und ohne Markierungen durch eine einzelne Kante ersetzt und
2. diese mit dem Wort beschriftet, das entlang dieses Pfades stand.

(»Patricia« steht für *»Practical Algorithm To Retrieve Information Coded in Alphanumerics«*.)

Beispiel: Patricia-Baum für die Methoden von `java.lang.Compiler`



3.2.4 Implementation

3-15

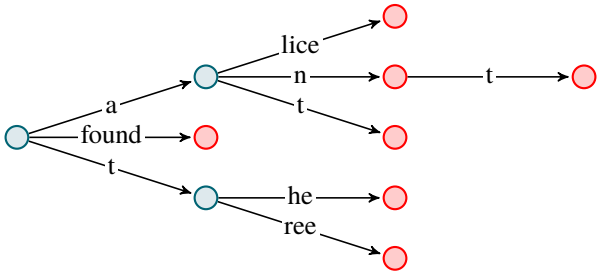
Implementationsdetails I

Speicherung von Patricia-Bäumen

Die in einem Trie gespeicherten Strings stehen oft auch noch an anderer Stelle im Speicher: Beispielsweise beim Trie der Worte eines Textes. In diesem Fall kann man in einem Patricia-Baum *nur die Indizes in den Text* speichern *statt dem Text selbst*.

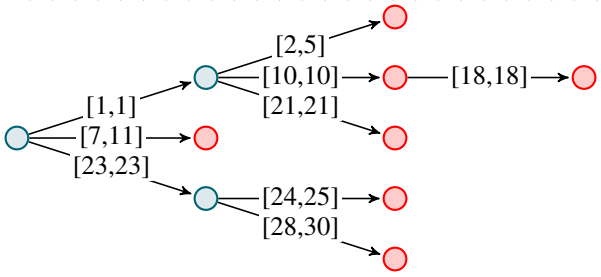
Beispiel: Patricia-Baum mit expliziten Kantenbeschriftungen

a	l	i	c	e		f	o	u	n	d		a	n		a	n	t		a	t		t	h	e		t	r	e	e
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30



Beispiel: Patricia-Baum mit impliziten Kantenbeschriftungen

a	l	i	c	e		f	o	u	n	d		a	n		a	n	t		a	t		t	h	e		t	r	e	e
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30



3-16

Implementationsdetails II

Speicherung der Kinder

Das Problem der vielen Kinder

Ein Trie hat in der Nähe der Wurzel einen *hohen Verzweigungsgrad*. Dort kann es für *jedes Symbol des Alphabets* ein Kind geben. Bei großen Alphabeten wie dem Unicode ist *unklar, wie man die Kinder effizient speichert*.

✎ **Zur Diskussion**

Geben Sie für jede der folgenden Arten der Speicherung der Kinder eines Knotens *Vorteile* und *Nachteile* an:

1. Als verkettete Liste
2. Als sortierter Array
3. Als balancierter Suchbaum
4. Als Hashtabelle

3.3 Suffix-Tries und -Trees

Das Problem: Die Suche in einem Virus-Genom

3-17

Wie haben bis jetzt Tries dafür genutzt, *ganze Worte* in einem langen Text schnell zu finden. Sucht man aber *in einem Virus-Genom* nach einem RNA-Schnipsel, so *gibt es keine »Wortgrenzen«.*

Beispiel

HIV-Genom
```
...atatttgctataaagaaaaaagacrgtact
aaatggagaaaattagtagatttcagagaa...
```
Schnipsel
```
acccgattattgga
```

3.3.1 Suffix-Tries

Die Lösung: Suffix-Tries.

3-18

Big Idea

Gegeben sei ein *langer Text ohne Wortgrenzen* (also ein langes Wort $w \in \Sigma^*$), in dem wir suchen wollen.

1. Betrachte die Menge S aller *Suffixe von* w.
2. Bilde den Trie dieser Menge.

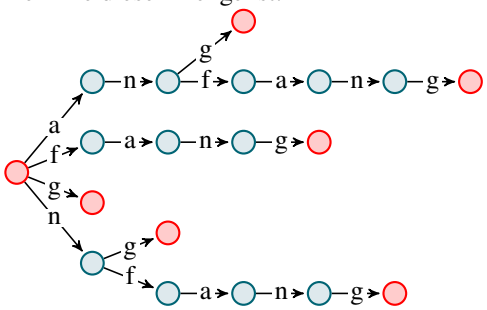
Diesen nennt man dann *Suffix-Trie* des Wortes.

Beispiel: Der Suffix-Trie des Wortes »anfang«

Die Menge S aller Suffixe von $w = anfang$ ist:

$$S = \{anfang, nfang, fang, ang, ng, g, \varepsilon\}$$

Der Trie dieser Menge ist:



Wie man mit Suffix-Tries Worte findet.

3-19

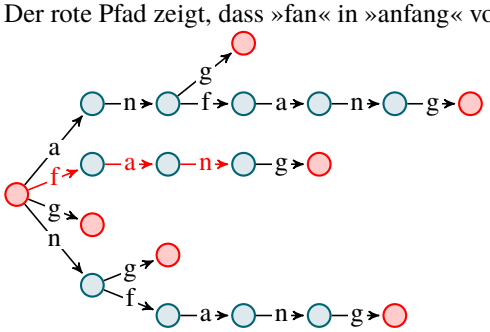
▶ **Lemma**

Sei T der Suffix-Trie von w. Ist v ein Teilwort von w, so gibt es in T eine Pfad beginnend bei der Wurzel, der genau mit v beschriftet ist.

Beweis. Kommt v in w vor, so ist v insbesondere der Anfang (Präfix) eines Suffix s von w. Folglich beginnt der Pfad in T, der zu s führt, mit v. □

Beispiel: »fan« kommt in »anfang« vor

Der rote Pfad zeigt, dass »fan« in »anfang« vorkommt.



3-20

✎ Zur Übung

Wie groß ist ein Suffix-Trie eines Wortes mit zehn Buchstaben

1. höchstens und
2. mindestens?

3.3.2 Suffix-Trees

3-21

Suffix-Trees sind Patricia-Varianten von Suffix-Tries

Suffix-Tries können *sehr groß werden*, sie sind dann aber auch *hochgradig redundant,* da sie viele lange, gleichartige Ketten enthalten.
Kodiert man Suffix-Tries als Patricia-Bäume, so ist dies *besonders effizient*.

▶ Definition

Ein *Suffix-Tree* eines Wortes w ist der Patricia-Baum der Menge aller Suffixe von w.

▶ Satz

Der Suffix-Tree eines nichtleeren Wortes w hat höchstens $2|w|-1$ Knoten.

Beweis. Ein Suffix-Tree hat höchstens $|w|$ Blätter, da jedes Blatt einem nichtleeren Suffix entspricht. Jeder Baum mit n Blättern hat höchstens $n-1$ innere Knoten vom Grad mindestens 2. Jeder Knoten in einem Patricia-Baum hat einen Grad von mindestens 2. Also gibt es höchstens $|w|-1$ innere Knoten und somit höchstens $2|w|-1$ Knoten insgesamt. □

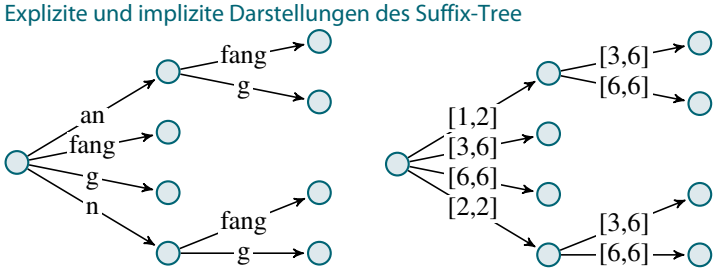
3-22

Der Suffix-Tree von »anfang«.

Das Wort

a	n	f	a	n	g
1	2	3	4	5	6

Explizite und implizite Darstellungen des Suffix-Tree



3-23

Zur Berechnung von Suffix-Trees.

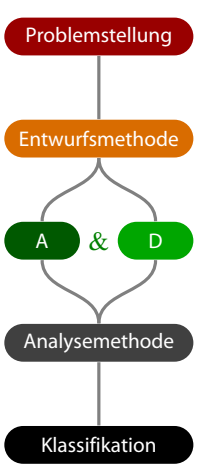
Offenbar kann man einen Suffix-Tree in *quadratischer Zeit* berechnen: Man fügt einfach alle Suffixe nacheinander ein.
Es geht jedoch wesentlich besser:

▶ Satz: Ukkonnen, 1995

Man kann den Suffix-Tree eines Wortes w in Zeit $O(|w|)$ berechnen.

Bemerkungen: Der Algorithmus ist zwar nicht übermäßig komplex, aber auch nicht ganz einfach. Die Hauptschwierigkeit liegt (mal wieder) darin, zu beweisen, dass er funktioniert. Hierzu sei auf den Artikel von Ukkonnen verwiesen.

Zusammenfassung dieses Kapitels



	Suchen in Wort-Mengen	Suchen in langen Texten
Entwurfsmethode	Teilen-und-Herrschen, Indizierung	Teilen-und-Herrschen, Indizierung
A & D	Tries	Suffix-Tries Suffix-Trees
Analysemethode	Laufzeitanalyse	Platzanalyse
Klassifikation	Suche nach v in $O(\|v\|)$	Größe $O(\|w\|^2)$ Größe $O(\|w\|)$

▶ Trie und Patricia-Baum

Ein *Trie* ist ein Suchbaum, in dem jeder Knoten für jedes Zeichen eines Alphabets maximal ein Kind hat. Ein *Patricia-Baum* ist eine kompakte Darstellung von Tries, in der Pfade zu einzelnen Kanten »komprimiert« sind.

▶ Suffix-Tries und -Trees

1. Ein *Suffix-Trie* ist der Trie der Suffixes eines Wortes.
2. Ein *Suffix-Tree* ist der Patricia-Baum des Suffix-Trie.

Zum Weiterlesen

[1] Esko Ukkonen, On-line construction of suffix trees, *Algorithmica*, 14(3):249–260, 1995.

Der in diesem Artikel vorgestellte Linearzeitalgorithmus zur Konstruktion von Suffix-Trees ist, zumindest was die Beschreibung angeht, einfacher als frühere Algorithmen und vor allen Dingen ist es ein *Online-Algorithmus:* Die Zeichen des Wortes werden von links nach rechts gelesen; frühere Algorithmen bauten den Suffix-Tree hingegen »von rechts nach links« auf. Der Artikel ist schön lesbar und mit vielen Beispielen ausgestattet.

Übungen zu diesem Kapitel

Bei den folgenden Übungen werden wir uns etwas genauer mit dem Zungenbrecher

In Ulm und um Ulm und um Ulm herum.

beschäftigen.

Übung 3.1 Konstruktion von Tries, leicht

Konstruieren Sie einen Trie, der genau die Wörter des obigen Zungenbrechers enthält. Zwischen Groß- und Kleinbuchstaben soll dabei nicht unterschieden werden.

Übung 3.2 Konstruktion von Suffix-Tries und -Trees, leicht

Wir betrachten die zusammengeschobene und vereinfachte Variante

u l m u n d u m u l m

des obigen Satzes. Konstruieren Sie zuerst den Trie aller Suffixe dieses Wortes und formen diesen dann in einen Suffix-Tree um.

Übung 3.3 Konstruktion von Suffix-Tries und Trees, mittel

Konstruieren Sie für den Text

f i s c h e r f i s c h e n f i s c h

zuerst den Trie aller Suffixe und formen diesen dann in einen Suffixtree um.

4-1

Kapitel 4
A&D: Indizieren mit Arrays

Von der Kunst, einen kompakten Index zu erstellen

4-2

Lernziele dieses Kapitels

1. Die Datenstruktur des Suffix-Arrays kennen
2. Den DC3-Algorithmus verstehen

Inhalte dieses Kapitels

Worum es heute geht

In diesem Kapitel wird es musikalisch, die Algorithmen tanzen zu Strings im Drei-Viertel-Takt. Anlass ist die Implementation des Difference-Cover-Modulo-3-Algorithmus, besser bekannt als DC3-Algorithmus, bei der an »taktvollen« Stellen im String andere Dinge passieren als an den eher »taktlosen«.

Ziel des Algorithmus ist die Berechnung eines Suffix-*Arrays,* neben Suffix-Tries und Suffix-Trees die dritte Art, Suffixe zu speichern. Suffix-Arrays sind konzeptionell sehr einfach (der Array der Startpositionen der Suffixes eines Strings in lexikographischer Reihenfolge), weshalb sie auch sehr gut und einfach in der Praxis nutzbar sind.

Genau wie bei Suffix-Trees ist das Hauptproblem bei Suffix-Arrays, sie effizient zu berechnen. Eine naive Implementation benötigt genau wie bei Suffix-Trees quadratische Zeit und dies ist bei Eingabegrößen von mehreren Megabytes inakzeptabel. Der tanzende DC3-Algorithmus aus diesem Kapitel meistert die Aufgabe hingegen in *linearer Zeit.* Im Vergleich entspricht DC3 daher einem sprintenden Eiskunstläufer, während die naiven Algorithmen zur Erstellung von Suffix-Arrays eher ein behäbiges Elefantenballett aufführen.

4.1 Suffix-Arrays

4.1.1 Suffix-Bäume sind gut. . .

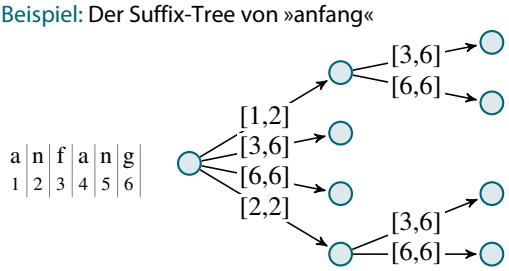
Ein Werbeplakat für Suffix-Trees.

4-4

Die zentralen Eigenschaften von Suffix-Trees

1. Der Suffix-Tree von w benötigt nur $O(|w|)$ Speicher.
2. Er kann in Zeit $O(|w|)$ konstruiert werden (Ukkonens Algorithmus).
3. Er erlaubt, in $O(|v|)$ Zeit nach einem Teilwort v von w zu suchen.

Beispiel: Der Suffix-Tree von »anfang«



4.1.2 . . . aber nicht gut genug

Suffix-Trees sind eine dolle Sache, *aber. . .*

4-5

Nachteile

- Die Speicherung der Kinder eines Knotens ist schwierig, insbesondere bei großen Alphabeten wie dem Unicode.
- Für jeden Knoten muss ein Objekt gespeichert werden, das wiederum mindestens zwei, eher noch mehr Verweise enthält.
- Algorithmen wie der von Ukkonen benötigen noch weitere Verweise in jedem Knoten.

Insgesamt benötigt ein Suffix-Tree pro Byte des zu speichernden Wortes etwa *12 bis 24 Byte an Speicher*.

Ziel

Eine Datenstruktur, die *»fast genauso gut«* wie ein Suffix-Tree ist, aber *»einfacher zu benutzen«* ist.

4.1.3 Die Idee

Suffix-Arrays sind einfach und kompakt

4-6

Die zwei Ideen hinter Suffix-Arrays

Sei w ein Wort und sei $w_i = w[i, \ldots, |w|]$ der an Stelle i beginnende Suffix von w.

1. Wir betrachten wieder die Menge aller Suffixe von w und *sortieren diese lexikographisch*. Dies ergibt »fast« den Suffix-Array.
2. Wir müssen die Suffixe aber nicht explizit speichern, es genügt, ihre *Anfangspositionen in w* zu speichern.

▶ **Definition:** Suffix-Array

Sei w ein Wort. Der *Suffix-Array* von w ist ein Array S von Indizes, so dass für jede zwei Indizes i und j mit $i < j$ gilt

$$w_{A[i]} < w_{A[j]}.$$

Hierbei steht $<$, wie auch im Folgenden, für die lexikographische Ordnung.

Beispiel: Suffix-Array von »anfang«

Das Wort »anfang« hat

1. die *Suffixe* »anfang«, »nfang«, »fang«, »ang«, »ng«, »g« und ε,
2. also sortiert: ε, »anfang«, »ang«, »fang«, »g«, »nfang«, »ng«.
3. Als Array von Anfangspositionen: $[7, 1, 4, 3, 6, 2, 5]$.

4.1.4 Vergleich von Suffix-Arrays und Suffix-Bäumen

4-7

Suffix-Arrays versus Suffix-Trees.

Suffix-Arrays sind fast so schnell wie Suffix-Trees:

- Zur Erinnerung: Der Suffix-Tree von w erlaubt es uns, für jedes Wort v in *Zeit* $O(|v|)$ zu testen, ob v in w vorkommt.
- Mit einem *Suffix-Arrays* geht dies *auch schnell:* Man führt einfach *eine binäre Suche* nach v durch, was $O(|v| \log |w|)$ dauert.

Suffix-Arrays brauchen viel weniger Platz:

- Offenbar benötigt ein Suffix-Array nur $\log_2 |w|$ Bits pro Zeichen des Arrays, also in der Praxis zwischen 1 und 4 Byte.
- Ein Suffix-Tree benötigt eher 12 bis 24 Byte pro Zeichen des Wortes.

Suffix-Arrays sind einfach:

- Der geringe Platzverbrauch erlaubt es, größere Teile des Arrays im Cache zu halten.
- Er kann als `short[]` oder `int[]` gespeichert werden, deren Verarbeitung sich besser optimieren lässt als die von Knoten-Objekten.

4-8

✎ **Zur Übung**

Für Anfänger Bestimmen Sie den Suffix-Array und den Suffix-Baum von »annanas«.

Für Profis Wie kann man mittels des Suffix-Array von w nach einem Wort v in Zeit $O(|v| + \log |w|)$ suchen statt lediglich in Zeit $O(|v| \log |w|)$?

4.2 Der DC3-Algorithmus

4.2.1 Das Ziel: ein Linearzeitalgorithmus

4-9

Wie schnell kann man Suffix-Arrays berechnen?

Ein Suffix-Array ist nichts anderes als die Sortierung der Suffixe von w, weshalb man ihn durch *jeden beliebigen Sortieralgorithmus* bestimmen kann. Man muss dabei *noch nicht einmal die Suffixe explizit hinschreiben*, sondern speichert nur die Anfänge.

Beispiel: Sortierung mit Bubble-Sort

»Unsortierter« Array

[	1	2	3	4	5	6	7	]
	anfang	nfang	fang	ang	ng	g	ε	

Erster Tausch

[	1	3	2	4	5	6	7	]
	anfang	fang	nfang	ang	ng	g	ε	

Zweiter Tausch

[	1	3	4	2	5	6	7	]
	anfang	fang	ang	nfang	ng	g	ε	

Dritter Tausch

[	1	4	3	2	6	5	7	]
	anfang	ang	fang	nfang	g	ng	ε	

Vierter Tausch

[	1	4	3	2	5	7	5	]
	anfang	ang	fang	nfang	ng	ε	ng	

Endresultat

[	7	1	4	3	6	2	5	]
	ε	anfang	ang	fang	g	nfang	ng	

Beobachtung
Nehmen wir an, wir nutzen *Merge-Sort* als Sortieralgorithmus. Dann werden $O(n \log n)$ *Vergleiche* durchgeführt. Der lexikalische Vergleich zweier Suffixe *kann nicht länger als* $O(n)$ dauern.
Hieraus ergibt sich: Man kann den Suffix-Array eines Wortes w in Zeit $O(n^2 \log n)$ berechnen mittels Merge-Sort.

Unser Ziel
Ein *Linearzeit-Algorithmus* für die Berechnung von Suffix-Arrays.

4.2.2 Der Algorithmus im Überblick

Wie man Suffix-Arrays in linearer Zeit berechnet. 4-10

▶ Satz: Kärkkäinen, Sanders, Burkhardt, 2003
Man kann den Suffix-Array eines Wortes w in Zeit $O(|w|)$ berechnen.

Der Algorithmus, genannt DC3, ist ein *Teile-und-Herrsche-Algorithmus*. Es gab vorher schon den Algorithmus von Farach, der grob so arbeitet:

- Teile die Stellen des Wortes in *gerade und ungerade Stellen* auf.
- Berechne rekursiv Suffix-Arrays hierzu.
- Setze diese durch einen sehr komplexen Merge-Schritt zusammen.

Neu am DC3-Algorithmus ist:

- Man halbiert nicht die Array-Größe, sondern
- reduziert die Größe nur auf zwei Drittel.
- Dadurch wird alles »ganz einfach«.

Überblick zum DC3-Algorithmus 4-11

▶ Definition: Takt
Sei w ein Wort. Wir nennen je drei aufeinanderfolgende Stellen in w eine *Takt*. Die jeweils erste Stelle eines Takts heißt *taktvoll*. Die anderen beiden Stellen heißen *taktlos*.
Fängt ein Suffix an einer taktvollen Stelle in w an, so nennen wir es taktvoll, sonst taktlos.

Beispiel
Das Wort »anfang« hat

- die taktvollen Suffixe »anfang«, »ang« und ε und
- die taktlosen Suffixe »nfang«, »fang«, »ng« und »g«.

Die drei Schritte des DC3-Algorithmus.

Erster Schritt: Sortierung der taktlosen Suffixe
Wir sortieren die Menge der taktlosen Suffixe. Dies ist (fast) die Bestimmung eines Suffix-Arrays, weshalb wir dies *rekursiv machen können.*
Beispiel: Wir sortieren rekursiv »nfang«, »fang«, »ng«, »g« zu »fang«, »g«, »nfang«, »ng«.

Zweiter Schritt: Sortierung der taktvollen Suffixe
Wir sortieren dann die Menge der taktvollen Suffixe. Dies geht ganz einfach, wenn man die Sortierung der taktlosen hat.
Beispiel: Wir sortieren rekursiv »anfang«, »ang«, ε zu ε, »anfang«, »ang«.

Dritter Schritt: Verschmelzung
Wir verschmelzen die beiden sortierten Listen wie bei Merge-Sort.

4.2.3 Schritt 1: Sortierung der taktlosen Suffixe

4-12

Superzeichen . . .

In dem Algorithmus werden wir öfter *Tripel von Zeichen* (also die Zeichen eines »Taktes«) als *ein Zeichen* behandeln.

▶ Definition: Superzeichen
Sei Σ ein Alphabet. Ein *Superzeichen* ist ein Zeichen aus dem Alphabet Σ^3.

Beispiel
Aus den drei Zeichen a, n und f wird das »Superzeichen« [anf], wobei die eckigen Klammern klar machen sollen, dass es sich um *ein* Zeichen handelt.

Sortierung von Zeichen und Superzeichen

- Wie setzen voraus, dass *unsere Alphabete sortiert sind*, es also eine lineare Ordnung auf Σ gibt.
- Diese überträgt sich auf *Superzeichen*, in dem diese »wie im Telefonbuch« sortiert werden: [aac] < [aba] < [abc] < [def].

4-13

. . . und Superstrings

Fasst man in einem String je drei Zeichen zu einem Superzeichen zusammen, so erhält man einen Superstring.

▶ Definition: Superstring
Sei $w \in \Sigma^{3n}$ ein Wort. Dann ist der *Superstring* zu $w \in (\Sigma^3)^n$ das Wort, das entsteht, wenn man je drei Zeichen von w zu einem Superzeichen zusammenfasst.
Ist die Länge von w nicht durch 3 teilbar, so ergänzen wir am Ende von w noch »Null-Zeichen«, bis dies der Fall ist.

Beispiel

- Aus dem Wort »anfang« wird der Superstring »[anf][ang]« der Länge 2.
- Aus dem Wort »fang« wird der Superstring »[fan][g00]« der Länge 2.

4-14

Die Alphabete von Superstrings

Ein Problem
Durch die *Rekursion* werden vom Algorithmus erst *Superstrings*, dann *Supersuperstrings* und später *Supersuper. . . superstrings* gebildet: Der ganze Faust ist irgendwann nur ein Zeichen. Dadurch entstehen *gigantische Alphabete* und man kann *nicht mehr zwei Supersuper. . . superzeichen in Zeit $O(1)$ vergleichen.*

Die Lösung
Normalerweise sind die Alphabete (wie ASCII oder Unicode) konstant. Der DC3-Algorithmus arbeitet hingegen *lieber mit dem Alphabet* $\{0,\ldots,|w|\}$, also mit »Zahlen« als »Zeichen«. Dies ist keine Einschränkung, denn ein String der Länge $|w|$ kann nicht mehr als $|w|$ Zeichen enthalten und wir können die vorhandenen Zeichen zunächst sortieren.

Beispiel
Den String aaxb $\in \Sigma^4$ kann man darstellen als $(1,1,3,2) \in \{0,\ldots,4\}^4$, da in dem String die Zeichen a, b und x vorkommen und diese sortiert sind: a < b < x.

Beispiel
Den Superstring [nfa][ng0][fan][g00] kann man darstellen als $(3,4,1,2) \in \{0,\ldots,4\}^4$, da gilt [fan] < [g00] < [nfa] < [ng0].

In den folgenden Beispielen werden trotzdem weiter Buchstaben verwendet, *intern* rechnet der Algorithmus aber mit Zahlen.

Erster Schritt des DC3-Algorithmus: Rekursion auf geeignete Superstrings.

4-15

Der rekursive Aufruf bei DC3

1. Bilde den Superstring R_1 von w ohne sein erstes Zeichen.
2. Bilde den Superstring R_2 von w ohne seine ersten beiden Zeichen.
3. Berechne (rekursiv) den Suffix-Array S des verketteten Superstrings R_1R_2.

Beispiel: Das Wort »anfang«

1. $R_1 =$ [nfa][ng0].
2. $R_2 =$ [fan][g00].
3. Der Suffix-Array von [nfa][ng0][fan][g00] lautet:
   - Die Suffixe sind [nfa][ng0][fan][g00], [ng0][fan][g00], [fan][g00], [g00] und ε.
   - Sortiert sind dies ε, [fan][g00], [g00], [nfa][ng0][fan][g00] und [ng0][fan][g00].
   - Also ist der Suffix-Array $[5,3,4,1,2]$.

Was den Superstring R_1R_2 so besonders macht.

4-16

▶ **Lemma**
Die Suffixe von R_1R_2 sind in derselben Reihenfolge wie die Suffixe von w an den taktlosen Stellen.

Suffixe von »anfang«:
ε
anfang
ang
fang
g
nfang
ng

Suffixe von [nfa][ng0][fan][g00]:
ε
[fan][g00]
[g00]
[nfa][ng0][fan][g00]
[ng0][fan][g00]

4.2.4 Schritt 2: Sortierung der taktvollen Suffixe

Die Ränge der Suffixe

4-17

▶ **Definition:** Rang eines Suffix
Der *Rang* eines Suffix gibt an, wie viele Suffixe des Wortes lexikographisch kleiner sind. Der *taktlose Rang* gibt an, wie viele taktlose Suffixe lexikographisch kleiner sind. Für den taktlosen Rang des bei Stelle i beginnenden Suffix schreiben wir r_i; und setzen $r_{n+1} = r_{n+2} = 0$.

Der Rang des i-ten Suffix ist damit die Stelle im Suffix-Array, an der i auftaucht.

Beispiel: Ränge der Suffixe von »anfang«

Buchstaben	a	n	f	a	n	g
Stelle	1	2	3	4	5	6
Rang	1.	5.	3.	2.	6.	4.

Beispiel: Taktlose Ränge der taktlosen Suffixe von »anfang«

Buchstaben	a	n	f	a	n	g
Stelle (i)	1	2	3	4	5	6
Taktloser Rang (r_i)		3.	1.		4.	2.

4-18

Taktlose Ränge von taktlosen Suffixen

Die taktlosen Ränge der taktlosen Suffixe von w ergeben sich aus dem Suffix-Array von R_1R_2 wie folgt:

1. Jedem taktlosen Suffix entspricht genau eine Stelle in R_1R_2.
2. Im Suffix-Array von R_1R_2 kommt diese Stelle an einer Position i vor.
3. Dann ist der gesuchte Rang gerade $i-1$.

Beispiel

Der Suffix-Array von »[nfa][ng0][fan][g00]« ist $[5,3,4,1,2]$. Hieraus ergibt sich:

Buchstaben	a	n	f	a	n	g
Stelle	1	2	3	4	5	6
Taktloser Rang		3.	1.		4.	2.

Beispielsweise gilt:

- Das taktlose Suffix »nfang« entspricht »[nfa][ng0][fan][g00]«, dem ersten Suffix von R_1R_2, welches an Position 4 vorkommt.
- Das taktlose Suffix »fang« entspricht »[fan][g00]«, dem dritten Suffix von R_1R_2, welches an Position 2 vorkommt.

4-19

Zweiter Schritt des DC3-Algorithmus: Sortierung der taktvollen Suffixe

Die *lexikographische Sortierung der taktvollen Suffixe* ermittelt man nun wie folgt:

1. Bilde für jede taktvolle Position i das Paar $(w[i], r_{i+1})$.
2. Sortiere die Paare mittels Radix-Sort in linearer Zeit.

Beispiel

Buchstaben	a	n	f	a	n	g
Stelle	1	2	3	4	5	6
Taktloser Rang		3.	1.		4.	2.
Paare	(a,3)			(a,4)		
Sortierung	(a,3)			(a,4)		

Beispiel

Buchstaben	a	n	f	a	e	n	g	e	r
Stelle	1	2	3	4	5	6	7	8	9
Taktloser Rang		4.	3.		1.	5.		2.	6.
Paare	(a,4)			(a,1)			(g,2)		
Sortierung	(a,1)			(a,4)			(g,2)		

4-20

Zusammenfassung bis hierher

Der Algorithmus hat bis jetzt Folgendes berechnet:

1. Eine Sortierung der taktlosen Suffixe untereinander.
2. Eine Sortierung der taktvollen Suffixe untereinander.
3. Die taktlosen Ränge der taktlosen Suffixe.

Ziel ist es nun, daraus die Sortierung aller Suffixe zu ermitteln.

4.2.5 Schritt 3: Verschmelzung

Das Konzept des Vorrangs

4-21

▶ Definition: Vorrang

Gegeben seien eine taktlose Stelle i und eine taktvolle Stelle t. In folgenden Fälle hat i *Vorrang*:

1. Stelle $i+1$ ist taktlos und $(w[i], r_{i+1}) < (w[t], r_{t+1})$.
2. Stelle $i+2$ ist taktlos und $(w[i], w[i+1], r_{i+2}) < (w[t], w[t+1], r_{t+2})$ gilt.

Anderenfalls hat t Vorrang.

Beispiel

Buchstaben	a	n	f	a	e	n	g	e	r
Stelle	1	2	3	4	5	6	7	8	9
Taktloser Rang		4.	3.		1.	5.		2.	6.

- Die taktlose Stelle 3 hat Vorrang vor der taktvollen Stelle 7, da $(f,a,1) < (g,e,6)$.
- Die taktlose Stelle 2 hat keinen Vorrang vor der taktvollen Stelle 7, da $(n,3) > (g,2)$.

Zentrale Eigenschaft des Vorrangs

4-22

▶ Lemma

Eine taktlose Stelle i hat genau dann Vorrang vor einer taktvollen Stelle t, wenn $w_i < w_t$.

Beweis. Wir zeigen zwei Richtungen:

1. Stelle i habe Vorrang vor t. Ist $i+1$ taktlos und $(w[i], r_{i+1}) < (w[t], r_{t+1})$ so ist entweder $w[i] < w[t]$ und damit auch $w_i < w_t$ oder $w[i] = w[t]$ und $r_{i+1} < r_{t+1}$ und somit ebenfalls $w_i < w_t$.
   Ist $i+2$ taktlos und $(w[i], w[i+1], r_{i+2}) < (w[t], w[t+1], r_{t+2})$, so argumentiert man genauso.
2. Es gelte $w_i < w_t$. Dann gibt es zwei Fälle: Ist $i+1$ taktlos, so ist entweder $w[i] < w[t]$ oder $w[i] = w[t]$ und $w_{i+1} < w_{t+1}$ und somit $r_{i+1} < r_{t+1}$. Damit hat i Vorrang.
   Ist hingegen $i+2$ taktlos, so sieht man mit einem ähnlichen Argument, dass $(w[i], w[i+1], r_{i+2}) < (w[t], w[t+1], r_{t+2})$ und somit i wieder Vorrang hat. □

Der DC3-Algorithmus

4-23

Schritt 3: Verschmelzung der Listen

Verschmelzung der Listen

Verschmelze die sortierten Listen L und T der taktlosen und der taktvollen Suffixe wie folgt:

```
1 M ← empty list
2 while neither L nor T is empty do
3   if first element of L has precedence (Vorrang) then
4     remove first element of L and append it to M
5   else
6     remove first element of T and append it to M
7
8 append remaining elements of L or T to M
```

4-24

Zwei Beispiele für die Verschmelzung in Aktion

Beispiel

Buchstaben	a	n	f	a	n	g
Stellen	1	2	3	4	5	6
Taktloser Rang		3.	1.		4.	2.

T: anfang, ang
L: fang, g, nfang, ng
M:

Vorrang »anfang« an Stelle 1 wegen $(a,n,1) < (f,a,4)$.

T: ang
L: fang, g, nfang, ng
M: anfang

Vorrang »ang« an Stelle 4 wegen $(a,n,2) < (f,a,4)$.

T:
L: fang, g, nfang, ng
M: anfang, ang

Kopiere den Rest.

T:
L:
M: anfang, ang, fang, g, nfang, ng

Fertig.

Beispiel

Buchstaben	a	n	f	a	e	n	g	e	r
Stelle	1	2	3	4	5	6	7	8	9
Taktloser Rang		4.	3.		1.	5.		2.	6.

T: aenger, anfaenger, ger
L: enger, er, faenger, nfaenger, nger, r
M:

Vorrang »aenger« an Stelle 4 wegen $(a,1) < (e,5)$.

T: anfaenger, ger
L: enger, er, faenger, nfaenger, nger, r
M: aenger

Vorrang »anfaenger« an Stelle 1 wegen $(a,4) < (e,5)$.

T: ger
L: enger, er, faenger, nfaenger, nger, r
M: aenger, anfaenger

Vorrang »enger« an Stelle 5 wegen $(e,5) < (g,2)$.

T: ger
L: er, faenger, nfaenger, nger, r
M: aenger, anfaenger, enger

Vorrang »er« an Stelle 8 wegen $(e,6) < (g,2)$.

T: ger
L: faenger, nfaenger, nger, r
M: aenger, anfaenger, enger, er

Vorrang »faenger« an Stelle 3 wegen $(f,a,1) < (g,e,6)$.

T: ger
L: nfaenger, nger, r
M: aenger, anfaenger, enger, er, faenger

Vorrang »ger« an Stelle 7 wegen $(g,2) < (n,3)$.

T:
L: nfaenger, nger, r
M: aenger, anfaenger, enger, er, faenger, ger

Kopiere den Rest.

T:
L:
M: aenger, anfaenger, enger, er, faenger, ger, nfaenger, nger, r

Fertig.

Zusammenfassung der Geschwindigkeit des DC3-Algorithmus.

4-25

Die Laufzeit $T(n)$ des Algorithmus bei Worten der Länge n errechnet sich wie folgt:

1. Die Bestimmung des Strings R_1R_2 im ersten Schritt benötigt Zeit $O(n)$. Die Rekursion benötigt Zeit $T(\frac{2}{3}n)$. Die Bestimmung der taktlosen Ränge benötigt wieder Zeit $O(n)$.
2. Die Sortierung der taktvollen Suffixe mittels Radix-Sort benötigt $O(n)$ Zeit.
3. Die Verschmelzung benötigt $O(n)$ Zeit.

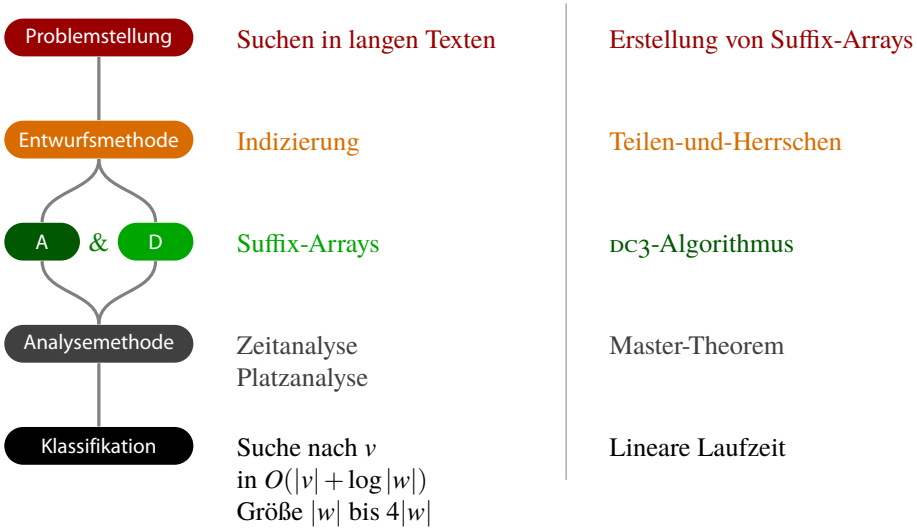
Dies liefert die *Rekursionsgleichung*

$$T(n) = T(\tfrac{2}{3}n) + \Theta(n).$$

Das Master-Theorem liefert (wegen Herrschaftsexponent $h = \log_{3/2}(1) = 0$ und Teilungsexponent $t = 1$), dass $T(n) = \Theta(n)$ gilt.

Zusammenfassung dieses Kapitels

4-26



▶ Suffix-Array

Der *Suffix-Array* von w ist der Array der Startpositionen der Suffixe von w in lexikographischer Reihenfolge.

▶ Satz

Der Suffix-Array von w lässt sich in Zeit $O(|w|)$ berechnen mittels des DC3-Algorithmus.

Zum Weiterlesen

[1] Juha Kärkkäinen, Peter Sanders, Stefan Burkhardt, Linear Work Suffix Array Construction, *Journal of the ACM*, 53(6):918–936, 2006.

Dieser Artikel beweist, dass man nicht immer hochkomplexe Mathematik braucht, um es in das Journal of the ACM zu schaffen – das »Nature« oder »Science« der Informatik. Der Beweis der Korrektheit des in dem Artikel vorgestellten DC3-Algorithmus passt auf drei (!) Zeilen und eine vollständige (!) Implementation des Algorithmus ist im original C++-Code auf 82 Zeilen angegeben. Tatsächlich ist der Artikel erschienen gerade *weil* der vorgestellte Algorithmus so einfach ist; insbesondere dann, wenn man ihn mit den komplexen Ideen früherer Verfahren vergleicht.

5-1

Kapitel 5
A&D: Textkompression

Fss dch krz nd knpp.

5-2

Lernziele dieses Kapitels

1. Zeichenbasierte Kompressionsalgorithmen kennen
2. Burrows-Wheeler-basierte Kompressionsalgorithmen kennen

Inhalte dieses Kapitels

Worum es heute geht

> ie Eei, a ee eie eie eua aueie, i i a eu. eiieeie i e i iee iee oioa, ie oae eia euae.

Nicht verstanden? Probieren Sie es mal hiermit:

> D rknntns, dss Txt n gwssn Rdndnz fwsn, st ncht gnz n. Bsplsws st s n vln Schrftsystmn ptnl, d Vkl nfch wgzlssn.

Die beiden Textbeispiele lehren uns einiges darüber, wie viel Information eigentlich in einem Text steckt. Wie man sieht, kann man bei vielen Worten einfach die Vokale weglassen und versteht noch immer recht gut, was gemeint ist (wie bei »Schrftsystmn«). Man könnte auch sagen, dass die eigentliche Information »in den Konsonanten« steckt. Viele Schriftsysteme machen sich dies zu Nutze und lassen die Vokale in der Regel ganz weg.

Die Textbeispiele lehren uns auch, dass die normale Art, einen Text zu speichern – nämlich als String, wo pro Zeichen ein oder zwei Byte spendiert werden – nicht besonders effizient ist. Man kann Texte *komprimiert* speichern, ohne dabei irgendetwas an der Bedeutung zu ändern.

Die Welt der Kompressionsalgorithmen ist ausgesprochen groß. Wenn Sie in Theoretischer Informatik gut aufgepasst haben, dann wissen Sie beispielsweise, dass Kolmogorov das Konzept des ultimativen Kompressionsalgorithmus eingeführt hat. Die Kolmogorov-Kompression

ist beweisbar die beste überhaupt mögliche – und leider können wir auch beweisen, dass sie nicht berechenbar ist (und damit vom praktischen Standpunkt eher weniger geeignet). Es gibt damit ein »ideales Verfahren«, das wir nie werden erreichen können; jedoch kann man sich alle paar Jahre einen neuen Algorithmus ausdenken, der ein bisschen näher an die Qualität einer Kolmogorov-Kompression herankommt.

Sie kennen sicherlich schon das Konzept der Huffman-Kodierung, bei der lediglich die Zeichen einzeln möglichst kurz aufgeschrieben werden. Jedoch nimmt die Huffman-Kodierung keine Rücksicht auf die Reihenfolge der Buchstaben und diese ist sicherlich nicht ganz unwichtig.

Wesentlich besser als die einfache Huffman-Kodierung sind wörterbuchbasierte Algorithmen wie der von Lempel, Ziv und Welch. Diese bauen während sie einen Text lesen eine Art »Wörterbuch« auf. Wenn ein Wort dann ein zweites Mal auftaucht, wird es nicht wieder komplett hingeschrieben, sondern nur ein Verweis auf die Stelle im Wörterbuch vermerkt. Dies kann, wenn das Wörterbuch nicht zu groß ist, sehr platzeffizient sein.

Eine erstaunliche Neuerung in der Welt der Kompressionsalgorithmen war zweifelsohne die Idee, eine Burrows-Wheeler-Transformation für die Kompression von Texten zu nutzen. Was das genau ist und wie das genau geht, wird hoffentlich im Laufe dieses Kapitels klar werden; es sei nur jetzt schon erwähnt, dass Suffix-Arrays eine entscheidende Rolle spielen werden.

5.1 Einführung

Manche Zeichen sind wichtiger als andere.

5-4

✎ Zur Diskussion

Welche Zeilen aus einem Stück verbergen sich hinter den folgenden Zeilen?

ae u, a! iooie,
uiee u eii,
U eie au eooie!
uau ui, i eie eü.

✎ Zur Diskussion

Und hinter diesen?

Hb nn, ch! Phlsph,
Jrstr nd Mdcn,
nd ldr ch Thlg!
Drchs stdrt, mt hßm Bmhn.

Die Ziel der Textkompression.

5-5

Die Idee

Ziel einer *Textkompression* ist es, einen *String* mit *möglichst wenig Bits* zu speichern.

Die Formalisierung

Sei Σ ein Alphabet (wie ASCII oder UNICODE). Ein *Kompressionsverfahren* ist ein Paar von Funktionen:

1. Der *Kompressionsfunktion* $K\colon \Sigma^* \to \{0,1\}^*$ und
2. der *Dekompressionsfunktion* $D\colon \{0,1\}^* \to \Sigma^*$.

Hierbei muss gelten $D(K(w)) = w$ für alle $w \in \Sigma^*$.

Beispiele von Programmen, die Kompressionsverfahren implementieren, sind `gzip`, `bzip2`, `rar` und viele andere.

5-6

Wünschenswerte Eigenschaften von Kompressionsverfahren.

1. Die Bitstrings $K(w)$ sollten *möglichst kurz* sein.
2. Die Bitstrings $K(w)$ sollten für *in der Praxis häufige* w besonders kurz sein.
3. Die Berechnung von K sollte schnell sein.
4. Die Berechnung von D muss schnell sein.

Beispiel: Vergleich von Kompressionsverfahren
Originaldatei ist `faust.txt` mit 203.693 Zeichen.

	`compress`	`gzip`	`bzip2`
Größe komprimierter Faust	88.817	82.922	64.257
Prozent von Originalgröße	44%	41%	32%
Zeit zum Komprimieren	7ms	25ms	32ms
Zeit zum Dekomprimieren	4ms	4ms	16ms

5.2 Zeichenweise Kompression

5.2.1 Die Idee

5-7

Zeichenweise Kompression: Die Idee

Ein besonders einfaches Kompressionsverfahren ist die *zeichenweise* Kompression: Für jedes Zeichen des Alphabets wählt man (möglichst geschickt) einen Bitstring aus. Dann schreibt man für die einzelnen Buchstaben eines Wortes die zugehörigen Bitstrings einfach hintereinander weg. Wenn man die Bitstrings geeignet gewählt hat, kann man aus der Bitfolge auch das Ursprungswort eindeutig rekonstruieren.

Beispiel
Sei $\Sigma = \{a, b, c\}$ und kodieren wir a als 0, weiter b als 10 und schließlich c als 11, so ergibt sich

Wort	Bitstring (komprimiertes Wort)
aa	00
bb	1010
aabc	001011

5.2.2 Huffman-Kodierung

5-8

Die Methode von Huffman

Big Idea
Buchstaben, die in einem Text besonders häufig vorkommen, sollten durch einen kurzen Bitstring kodiert werden.

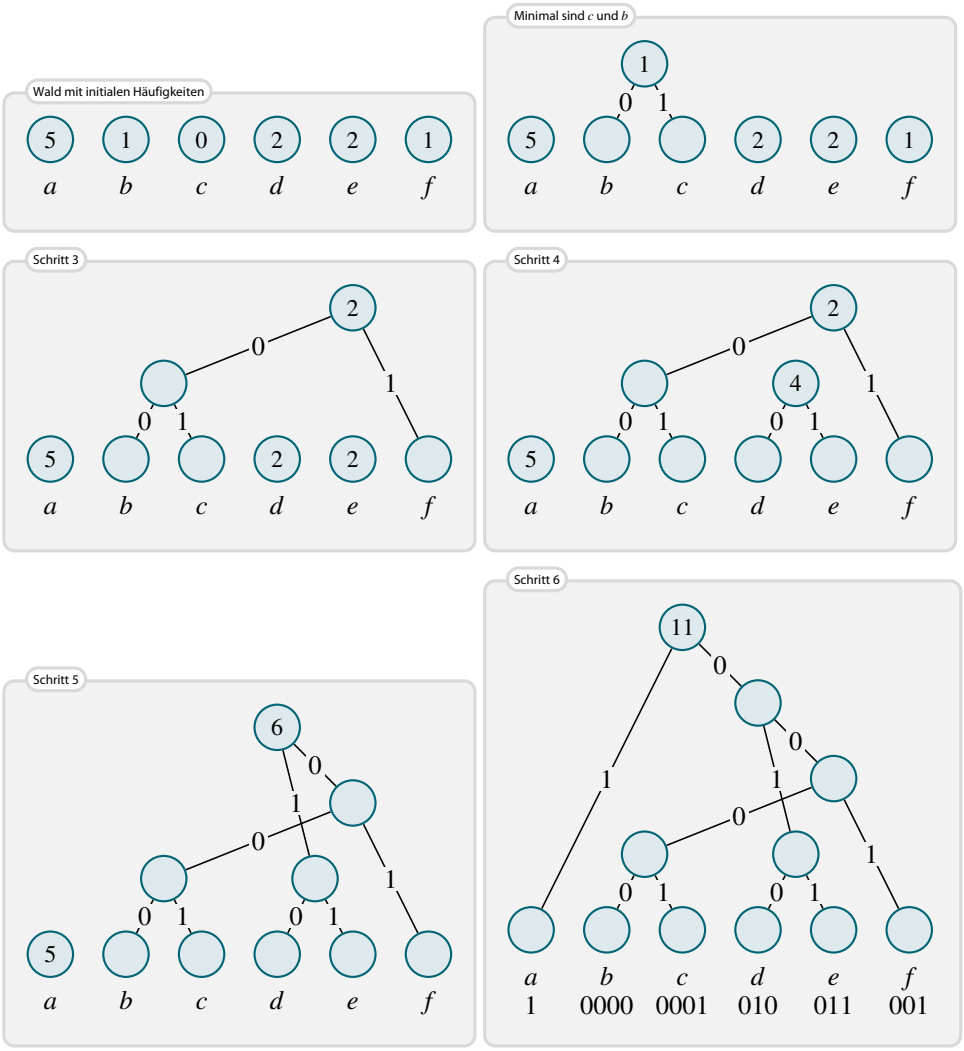
Die Methode von Huffman

1. Ermittle für jedes Symbol $\sigma \in \Sigma$, wie häufig es in w vorkommt.
2. Bilde einen »Wald« wie folgt: Für jedes Symbol $\sigma \in \Sigma$ füge einen »Baum« ein, der nur aus einem Knoten besteht und dessen Wurzel mit der Häufigkeit beschriftet ist.
3. Wiederhole nun Folgendes, so lange es noch mehr als einen Baum im Wald gibt:
   3.1 Finde zwei Bäume mit minimalen Beschriftungen an den Wurzeln.
   3.2 Fasse diese zu einem Baum zusammen, indem ein neuer Wurzelknoten hinzugefügt wird, der mit der Summe der Beschriftungen beschriftet wird.
   3.3 Beschrifte eine der neuen Kanten mit 0 und die andere mit 1.
4. Der Bitstring für ein Zeichen σ ist nun die Folge der Bits entlang des Wegs von der Wurzel zu dem Symbol.

Beispiel der Berechnung einer Huffman-Kodierung

5-9

Sei $\Sigma = \{a,b,c,d,e,f\}$ und $w = afaadeaadeb$.



$K(w) = 100111010011110100110000$ (plus die Kodierungstabelle)

Wie gut ist die Huffman-Kodierung?

5-10

Man kann zeigen:

▶ Satz

Unter allen zeichenweisen Kodierungen eines Wortes, bei denen kein kodierender Bitstring Präfix eines anderen ist, liefert die Huffman-Kodierung die kürzeste Kodierung.

5.2.3 Move-To-Front-Kodierung

Wie man die Häufigkeit von Buchstaben erhöht

5-11

Das Problem

- Betrachten wir das Wort

$$abababcdcdcdcd$$

- Hier kommen *alle Buchstaben gleichhäufig vor*, weshalb die *Huffman-Kodierung nichts bringt*.

Das Ziel

- Wir würden gerne das Wort schreiben als

$$12121212^*12121212$$

wobei das Sternchen dafür steht »ab hier ändern 1 und 2 ihre Bedeutung«.

– Jetzt kommen 1 und 2 sehr oft vor, das Sternchen hingegen nur einmal, weshalb sich das Wort gut Huffman-kodieren lässt.

5-12

Die Move-To-Front-Kodierung

Big Idea

– Bilde eine *Tabelle aller Zeichen* von Σ.
– Kodiere ein Zeichen als *seine Position in der Tabelle*, …
– …und bewege es dann an den Anfang der Tabelle.

Beispiel

		Tabelle			
Wort	Kodierung	1	2	3	4
ababcdcd		a	b	c	d
ababcdcd	1	a	b	c	d
abababcdcd	12	b	a	c	d
ababbcdcd	122	a	b	c	d
ababcdcd	1222	b	a	c	d
ababcdcd	12223	c	b	a	d
ababcdcd	122234	d	c	b	a
ababcdcd	1222342	c	d	b	a
ababcdcd	12223422	d	c	b	a

5.3 Kompression von Zeichenwiederholungen

5.3.1 Die Idee

5-13

Die Idee der Kompression von Zeichenwiederholungen

Big Idea

Das Wort *Tooooooooooooor*! lässt sich kodieren als $To^{14}r$!.

Bei einer *Kompression von Zeichenwiederholungen* wird eine häufige Wiederholung eines einzelnen Zeichens ersetzt durch die Anzahl der Wiederholungen gefolgt von dem Zeichen.

5.3.2 Run-Length-Kompression

5-14

Varianten von Run-Length-Kompression.

– Eine *Run-Length-Kompression* komprimiert Zeichenwiederholungen.
– Da sich aber *viele Zeichen* gar nicht wiederholen, muss man auch »Nicht-Wiederholung« effizient kodieren können.
– Die Verfahren unterscheiden sich darin, wie dies geschieht.

Variante 1: Escape-Zeichen

– Man führt ein spezielles »Escape«-Zeichen ein.
– Nur wenn diese Zeichen kommt, so folgt ein Zeichen und eine Anzahl von Wiederholungen.

Beispiel

Aus *Tooooooore*!!!! wird $T^{\text{ESC}}o6re^{\text{ESC}}!4$.

Variante 2: Sonderzeichen für die Anzahl

– Man führt *zwei* neue Symbole ein: Die Sonder-Null ($\bar{0}$) und die Sonder-Eins ($\bar{1}$).
– Folgt einem Zeichen eine Folge von Sonder-Nullen und -Einsen, so *gibt diese Folge den Binärcode der Anzahl der Wiederholung an*.
– Da so eine Folge immer mit einer Sonder-Eins anfangen würde, lässt man diese weg.

Beispiel

Aus *Tooooooor*!!!! wird $To\bar{1}\bar{0}re!\bar{0}\bar{0}$, denn $\bar{1}\bar{0}$ steht für $6 = 110_2$ und $\bar{0}\bar{0}$ steht für $4 = 100_2$.

Dieses Verfahren ist besonders in Verbindung mit einer Huffman-Kodierung sinnvoll.

5.4 Suffixbasierte Kompression

5.4.1 Die Idee

Suffixbasierte Kompression: Die Idee

5-15

Die Probleme

- Zeichenbasierte Kodierungen nehmen *keine Rücksicht auf die Reihenfolge* der unterschiedlichen Zeichen.
  So ist »informatik« viel wahrscheinlicher als »amniifktr«; nach »anfan« kommt viel eher ein »g« als ein »x«.
- Bei der Kompression von Wiederholungen muss genau ein Zeichen sich wiederholen. Sie bringt nichts bei *abababababab*.

Die Lösung

- Man untersucht das Wort *global*, indem wir seinen *Suffix-Array* berechnen.
- Mittels des Suffix-Arrays *ändern wir die Reihenfolge der Buchstaben des Wortes*.
- Das neue Wort *enthält dieselbe Information wie das Ursprungswort und hat dieselbe Länge,* es lässt sich aber *besser komprimieren*.

5.4.2 Burrows-Wheeler-Transformation

Die Burrows-Wheeler-Transformation wirbelt die Buchstaben eines Wortes durcheinander.

5-16

▶ **Definition**

Sei $w \in \Sigma^*$ ein Wort. Die *Burrows-Wheeler-Transformation* von w ist ein um einen Buchstaben längeres Wort $\mathrm{BWT}(w)$, das wie folgt bestimmt werden kann:

1. Bilde alle Suffixe von w.
2. Sortiere diese lexikographisch.
3. Ersetze jedes Suffix durch den Buchstaben, der *in w direkt vor diesem Suffix kommt*.

Beispiel: Die Burrows-Wheeler-Transformation von $w = anfang$

1. Die Suffixe	2. Sortierung	3. Ersetzung
anfang	ε	*g*
nfang	*anfang*	^
fang	*ang*	*f*
ang	*fang*	*n*
ng	*g*	*n*
g	*nfang*	*a*
ε	*ng*	*a*

Also ist $\mathrm{BWT}(anfang) = g\text{^}fnnaa$.

✎ **Zur Übung**

5-17

Berechnen Sie $\mathrm{BWT}(annanas)$.

Erste Beobachtungen: Die Burrows-Wheeler-Transformation lässt sich gut komprimieren

5-18

- Aus *anfang* wird *g^fnnaa*.
- Hier kommen nun *dieselben Buchstaben mehrfach hintereinander* vor, was für eine *Run-Length-Kompression* besonders nützlich ist.
- Das *ist kein Zufall*, denn wenn ein Teilwort wie »häufig« in einem Text mehrfach vorkommt, so wird *vor allen mit »äufig« anfangenden Suffixe (fast) immer ein »h« stehen.*

5-19

Zweite Beobachtungen: Die Burrows-Wheeler-Transformation ist umkehrbar.

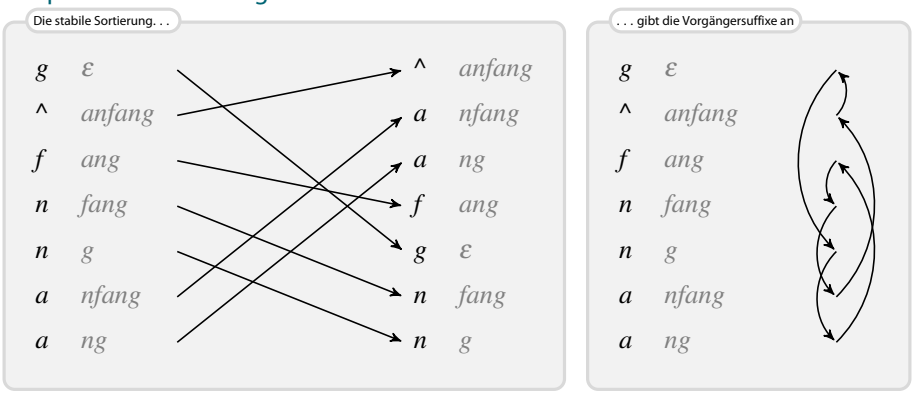
Hinter der *Umkehrung* der Burrows-Wheeler-Transformation stecken folgende Ideen:

Big Idea

- Man stelle sich die Buchstaben von $\mathrm{BWT}(w)$ als Spalte vor.
- Jede Zeile ist nun der Anfang eines Suffix.
- *Sortiert* man die Zeilen *stabil*, so sind die Zeilen nun gerade die Anfänge der *sortierten Suffixe*.
- *Daher gibt die Sortierreihenfolge für jedes Suffix an, welches Suffix sein Vorgänger in w ist.*

5-20

Beispiel der Umkehrung der Burrows-Wheeler-Transformation



5-21

Geschwindigkeit der Burrows-Wheeler-Transformation

▶ **Satz**

Sowohl die Burrows-Wheeler-Transformation wie auch ihre Umkehrung lassen sich in Zeit $O(n)$ berechnen.

Beweis. Das Wort $\mathrm{BWT}(w)$ ergibt sich unmittelbar aus dem Suffix-Array von w. Dieser kann aber mit dem DC3-Algorithmus in Zeit $O(n)$ berechnet werden.
Um die Umkehrung zu berechnen, muss man die Buchstaben von $\mathrm{BWT}(w)$ stabil sortieren, was mittels Radix-Sort in Zeit $O(n)$ möglich ist, und danach die ermittelte Permutation einmal durchlaufen. □

5.4.3 Seward-Kompression

5-22

Die Kompression nach Seward in `bzip2`.

Das Programm `bzip2` führt (etwas vereinfacht) folgende Schritte zur Kompression eines Wortes w durch:

1. Berechne die Burrows-Wheeler-Transformation von w.
2. Kodiere diese neu mittels einer Move-To-Front-Kodierung.
3. Führe darauf eine Run-Length-Kompression mittels Sonderzeichen für Anzahlen durch.
4. Führe eine Huffman-Kodierung durch.

Die Dekomprimierung invertiert all diese Schritte.

5.5 *Wörterbuchbasierte Kompression

5.5.1 Die Idee

Wörterbuchbasierte Kompression: Die Idee

5-23

Big Idea

Statt *einzelnen Zeichen* kodieren wir *größere Teilworte*. Diese Teilworte stehen in einem *Wörterbuch*. Ein Wort wird dann kodiert als *Folge von Positionen der Teilworte in dem Wörterbuch*. Das Wörterbuch ist *nicht fest*, sondern *wird dynamisch in Abhängigkeit des Textes gebaut.*

5.5.2 Lempel-Ziv-Welch-Kompression

Die Methode von Lempel-Ziv-Welch

5-24

Die Methode von Lempel, Ziv und Welch beruht auf zwei Ideen:

Das Wörterbuch

Das Wörterbuch ist ein *dynamisch wachsender Array A* von Worten über Σ (also grob ein `Array<String>`). Anfangs enthält der Array für jedes Symbol genau einen Eintrag.

Die Kodierungspärchen

Ein *Kodierungspärchen* ist ein Paar (i, σ), wobei

- i der Index eines Wortes $A[i]$ ist und
- σ ein Symbol ist.

Die *Bedeutung* eines solchen Pärchens ist:

1. Seine Dekodierung ist $A[i]\sigma$ (also $A[i]$ gefolgt von σ).
2. Das Wörterbuch wird um den neuen Eintrag $A[i]\sigma$ erweitert.

Beispiel eine Lempel-Ziv-Welch-Kompression

5-25

Sei $\Sigma = \{a, b, c, d, e, f\}$ und $w = afaadeaadeb$.

Anfangswörterbuch

i	0	1	2	3	4	5
A[i]	a	b	c	d	e	f

Wort: afaadeaadeb

Kodierungspärchen: noch keine

af neu im Wörterbuch

i	0	1	2	3	4	5	6
A[i]	a	b	c	d	e	f	af

Wort: afaadeaadeb

Kodierungspärchen: $(0, f)$

aa neu im Wörterbuch

i	0	1	2	3	4	5	6	7
A[i]	a	b	c	d	e	f	af	aa

Wort: afaadeaadeb

Kodierungspärchen: $(0, f)(0, a)$

de neu im Wörterbuch

i	0	1	2	3	4	5	6	7	8
A[i]	a	b	c	d	e	f	af	aa	de

Wort: afaadeaadeb

Kodierungspärchen: $(0,f)(0,a)(3,e)$

aad neu im Wörterbuch

i	0	1	2	3	4	5	6	7	8	9
A[i]	a	b	c	d	e	f	af	aa	de	aad

Wort: afaadeaadeb

Kodierungspärchen: $(0,f)(0,a)(3,e)(7,d)$

eb neu im Wörterbuch

i	0	1	2	3	4	5	6	7	8	9	10
A[i]	a	b	c	d	e	f	af	aa	de	aad	eb

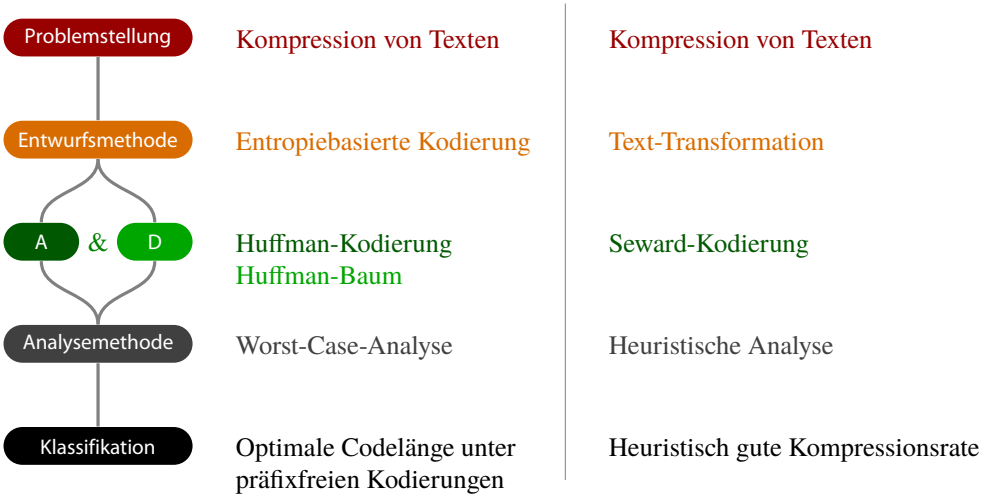
Wort: afaadeaadeb

Kodierungspärchen: $(0,f)(0,a)(3,e)(7,d)(4,b)$

Die Kodierungspärchen werden dann jeweils noch mit möglichst wenig Bits kodiert. Bei ASCII beispielsweise: $\lceil \log_2(\text{aktuelle Größe von } A) \rceil + 8$ Die so kodierten Pärchen können dann einfach hintereinanderweg stehen.

Zusammenfassung dieses Kapitels

5-26



- **Huffman-Kodierung**
  Ersetze jedes Zeichen eines Textes durch die Bits entlang des Pfades im Huffman-Baum.
- **Move-To-Front-Kodierung**
  Ersetze jedes Zeichen eines Textes durch seine Position in einer Tabelle und bewege es an den Anfang der Tabelle.
- **Kompression von Zeichenwiederholungen (Run-Length-Kompression)**
  Führe neue Zeichen $\bar{0}$ und $\bar{1}$ ein und schreibe statt den Wiederholungen nur deren Anzahl binär mittels $\bar{0}$ und $\bar{1}$ auf (lasse die führende $\bar{1}$ weg).

▶ Burrows-Wheeler-Transformation
Bilde den Suffix-Array eines Wortes w und ersetze jedes Suffix durch das Zeichen, das in w vor diesem Suffix kommt.

▶ Seward-Kodierung (bzip2)
Bei Eingabe w tue:

1. Wende die Burrows-Wheeler-Transformation an.
2. Wende Move-To-Front an.
3. Wende eine Run-Length-Kompression an.
4. Wende die Huffman-Methode an.

(Tatsächlich führt `bzip2` noch eine überflüssige Run-Length-Kompression vorneweg aus und benutzt mehrere Huffman-Bäume gleichzeitig.)

Übungen zu diesem Kapitel

Übung 5.1 Länge einer Run-Length-Kompression, mittel

Das Wort w habe die Form $a^n b^m c^p d^q$, wobei $a,b,c,d \in \Sigma$ und $n,m,p,q \geq 2$ gelte.

1. Wie lang ist die Run-Length-Kompression von w bei der Variante mit Sonder-Null und Sonder-Eins (das Alphabet ist also $\Sigma = \{a,b,c,d,\bar{0},\bar{1}\}$?
2. Wie viele Bits benötigt die Huffman-Kodierung (ohne Kodierungstabelle) dieser Run-Length-Kompression, wenn die Sonder-Null und die Sonder-Eins ähnlich oft vorkommen?
3. Wie viele Bits benötigt die Huffman-Kodierung, wenn die Zahlen n, m, p und q alles Zweier-Potenzen sind?

Übung 5.2 Effekt einer Move-To-Front-Kodierung, einfach

Das Wort w habe die Form $a^n b^m c^p a^q$, wobei $a,b,c \in \Sigma$ und $n,m,p,q \geq 2$ gelte. Geben Sie die Move-To-Front-Kodierung von w an.

In den folgenden Übungen geht es um die Frage, wie gut verschiedene Kompressionsverfahren Worte komprimieren, die nur aus Wiederholungen eines kurzen Wortes bestehen. Beispielsweise besteht *abababababab* aus fünf Wiederholungen von *ab*. Formal geht es also um Worte w der Form u^n (das bedeutet »w besteht aus n vielen Wiederholungen von u«), wobei die Länge m von u mindestens 2 sein soll und alle Zeichen von u sollen unterschiedlich sein. Das Wort w hat also Länge mn und wir stellen uns vor, dass n viel größer als m ist.

Übung 5.3 Burrows-Wheeler-Transformation sich wiederholender Worte, mittel

Sei $w = u^n$ mit $|u| = m > 1$ und alle Buchstaben in u seien unterschiedlich. Geben Sie die Burrows-Wheeler-Transformation von w zunächst für ein Beispiel und dann allgemein an.

Tipp: Führen Sie eine Bezeichnung für die Liste der in u vorkommenden Buchstaben in sortierter Reihenfolge ein.

Übung 5.4 Kodierungslänge sich wiederholender Worte, schwer

Sei $w = u^n$ mit $|u| = m > 1$ und alle Buchstaben in u seien unterschiedlich.

1. Wie viele Bits (ohne Kodierungstabelle) benötigt die Huffman-Kodierung von w?
2. Wie viele Bits (hier genügt die O-Klasse) benötigt die Seward-Kompression von w? Benutzen Sie dazu die Ergebnisse von Übungen 5.3, dann von 5.2 und dann von 5.1.
3. Überprüfen Sie Ihr Ergebnis, indem Sie eine Datei, die aus einer Million mal *ab* besteht, mittels `bzip2` komprimieren.

Teil III
Online-Probleme

Um ein mögliches Missverständnis gleich von vornherein auszuräumen: Das »Online« in »Online-Probleme« hat nicht mit »online« zu tun. Wenn heutzutage von »online« die Rede ist, dann denken wir unwillkürlich an Internet-Zugänge, das World-Wide-Web oder soziale Netzwerke wie Facebook. Über die Jahre erweitert sich der Begriff immer weiter – aus dem Web ist erst das Web 2.0 geworden und nun gerade das Web 3.0; das Wort »Facebook« kennt meine Rechtschreibkorrektur nicht, in ein paar Jahren ist das sicherlich anders – jedoch denkt kaum jemand bei dem Begriff »online« an das, worum es in diesem Teil der Veranstaltung gehen soll: Um eine temporale Eigenschaft von Eingabedaten. Insofern ist die Bezeichnung sicherlich nicht ganz glücklich gewählt (wie so viele Bezeichnungen in der Theoretischen Informatik), jedoch muss zu ihrer Verteidigung gesagt werden, »dass sie zuerst da war«: Online-Probleme wurden bereits von Theoretikern untersucht als Tim Berners-Lee noch an seinem Enquire-Programm herumbastelte und noch niemand »online war«.

Bei Online-Problemen geht es darum, dass am Anfang der Berechnung die Eingaben noch gar nicht vorliegen. Vielmehr muss man, ganz wie im richtigen Leben, Entscheidungen treffen aufgrund der bisher vorliegenden Informationen und dann hoffen, dass die später eintreffenden Daten diese Entscheidungen rechtfertigen. Wenn einige Entscheidungen getroffen wurden, kommen zusätzliche Daten, so dass man sich wieder entscheiden muss, und so fort.

Ein klassisches Beispiel eines Online-Problems ist das Caching-Problem: Bekanntermaßen ist im Hauptspeicher eines Rechners nicht genug Platz, um den unglaublichen Bedarf an virtuellem Speicher von allen Prozessen eines Systems zu befriedigen. Die Lösung ist, dass eben nur ein kleiner Teil des virtuellen Speichers real im Hauptspeicher liegt. Solang der Prozessor nur auf dieses Teil zugreift, ist alles prima. Wenn aber dann ein Datum aus dem virtuellen Speicher benötigt wird, das noch nicht im Hauptspeicher liegt, so muss dieses geladen werden und dafür ein anderes Datum im Hauptspeicher überschrieben werden. Die spannende Frage ist nun, welches Datum man überschreiben sollte. Idealerweise natürlich eines, das nie wieder oder zumindest möglichst lange nicht mehr gebraucht werden wird. Genau dies ist aber eine temporale Eigenschaft der Daten und wir wissen leider nicht, welche Daten in der Zukunft gebraucht werden und welche nicht.

Naturgemäß wird das Caching-Problem erst dadurch wirklich schwierig, dass eben nicht bekannt ist, welche Daten in der Zukunft benötigt werden. Wüsste man dies (man spricht dann von der »Offline-Variante«), so ist die Sache wesentlich einfacher und man könnte auch bessere Ergebnisse erzielen. Da Theoretiker, insbesondere Komplexitätstheoretiker, gerne alles und jedes vermessen, wüssten sie auch gerne, um wie viel genau die Offline-Variante leichter ist als die Online-Variante. Es würde nahe liegen, den Faktor, um den ein Offline-Verfahren besser ist als ein Online-Verfahren vielleicht den »Offline-Faktor« oder auch den »Online-Faktor« zu nennen – jedoch ist die offizielle Bezeichnung »kompetitive Rate«. Wie bereits erwähnt: Theoretiker sind nicht besonders gut darin, Namen zu vergeben.

6-1

Kapitel 6

Klassifikation: Kompetitive Rate

Was bringt es, wenn man in die Zukunft schauen kann?

Lernziele dieses Kapitels

1. Varianten von Scheduling- und Caching-Problemen kennen
2. Greedy-Verfahren zu deren Lösung kennen
3. Konzept des Online-Algorithmus verstehen
4. Konzept der kompetitiven Rate verstehen
5. Kompetitive Raten von Algorithmen bestimmen können

6-2

Inhalte dieses Kapitels

Worum es heute geht

Kann man in die Zukunft sehen? Es gibt zumindest genügend Leute, die behaupten, dass sie es können (Nostradamus, Wahrsager, Börsenhändler). Selbst die seriöse Wissenschaft hat sich dieser Frage bereits angenommen: Seit geraumer Zeit versucht man mittels so genannter Psi-Experimente zu klären, ob wir Menschen die Gabe haben, wenigsten eine bisschen in die Zukunft zu sehen.

Ein besonders interessantes Ergebnis hierzu stammt aus dem Jahr 2010 von Daryl Bem, der einen Artikel mit dem Titel *Feeling the Future: Experimental Evidence for Anomalous Retroactive Influence on Cognition and Affect,* erschienen im *Journal of Personality and Social Psychology,* veröffentlicht hat. Die para-wissenschaftliche Literatur ist voll von solchen Untersuchungen; das Besondere an dieser Arbeit ist, dass sie in einer anerkannten wissenschaftlichen Fachzeitschrift erschienen ist und dass sie den üblichen Standards genügt, die an psychologische Experimente gestellt werden.

Hier die Ergebnisse leicht (aber wirklich nur leicht) verkürzt zusammengefasst: Bei binären Entscheidungsexperimente wurde ein statistisch signifikanter Effekt gemessen, nach dem insbesondere Männer etwas in die Zukunft sehen können, wenn man ihnen als Belohnung pornographische Bilder in der nahen Zukunft zeigt.

Wie man sich vorstellen kann, hat dieses Ergebnis neben einiger Belustigung auch eine Menge an wissenschaftlichen Einwänden hervorgebracht. Die Quintessenz ist, dass die statistischen Methoden, die in dem Artikel genutzt wurden, um die vermeintliche Vorhersage der Zukunft zu beweisen, schlecht sind. Jedoch sind dies genau dieselben Methoden, die auch in vielen anderen Kontexten genutzt werden, beispielsweise wenn es um das Risiko des Passiv-Rauchens geht. Der Artikel beweist damit eigentlich nur, dass man solche Methoden viel kritischer sehen muss als dies üblich ist. Womit sich wieder der alte Spruch bewahrheitet: »Traue keiner Statistik, die du nicht selbst gefälscht hast.«

Was hat das aber alles mit Computern zu tun? In diesem Kapitel geht es um die Frage,

was es Computern helfen würde, wenn sie in die Zukunft sehen könnten. Wir untersuchen dazu *Online-Algorithmen*, welche immer wieder Entscheidungen fällen müssen, die sich in der Zukunft als eher gut oder doch als weniger gut herausstellen. Ein klassisches Beispiel sind die von modernen Betriebssystemen genutzten *Caching-Algorithmen,* die sich im Falle eines vollgelaufenen Hauptspeichers überlegen müssen, welche Teile des Hauptspeichers sie auf die Festplatte auslagern sollten. Idealerweise sollte man natürlich nicht genau die Teile auslagern, die man gleich wieder braucht; aber dafür müsste man eben in die Zukunft sehen können. Um wie viel besser Cache-Algorithmen würden, wenn sie dies könnten, wird durch die *kompetitive Rate* gemessen.

Was lehren uns die Untersuchungen von Bem in Bezug auf die kompetitive Rate von Caching-Algorithmen? Vielleicht könnte man Caching-Algorithmen mit einer besseren kompetitiven Rate bauen, wenn man ihnen eine »geeignete Belohnung« bietet. An pornographischem Material herrscht im Internet ja bekanntlich kein Mangel. Zu klären wäre allerdings, worauf Caching-Algorithmen wirklich stehen.

6.1 Einführung zu Online-Algorithmen

6.1.1 Das Ski-Leihen-Problem

6-4

Fahren Sie gerne Ski?

Das Ski-Leihen-Problem

Sie beschließen, diesen Winter mal wieder Ski zu fahren. Die Skier aus dem letzten Jahr können Sie nicht wieder nutzen, da diese vollkommen unmodisch sind. Es stellt sich die Frage, ob Sie für diese Saison neue Skier *kaufen oder leihen* sollten: Sie zu kaufen kostet Sie einmalig 200 Euro, sie zu leihen kostet Sie 10 Euro am Tag. Was sollten Sie tun?

6-5

Das Ski-Leihen-Problem als Online-Problem.

▶ Definition: Online-Problem

Bei einem *Online-Problem* erhält ein Algorithmus schrittweise Zugriff auf die Elemente einer *Folge von Ereignissen*. Nach jedem Ereignis muss der Algorithmus eine (oder mehrere) *Entscheidungen* treffen. Das Problem kann *Bedingungen* angeben, wann die Entscheidungen *korrekt* sind (wenn auch nicht unbedingt »gut«). Eine *Maßfunktion* gibt am Ende an, wie gut die Entscheidungen waren.

Beispiel: Das Ski-Leihen-Problem als Online-Problem

Ereignisse »Heute brauche ich Skier« und »Saison zu Ende«.

Entscheidungen »Für einen Tag leihen«, »kaufen« oder »nichts tun«.

Bedingungen
1. Das »Saison zu Ende« Ereignis kommt genau einmal am Ende.
2. Die Entscheidung »nichts tun« ist erst nach einer »kaufen« Entscheidung möglich.

Maß Die Anzahl der »Für einen Tag leihen« Entscheidungen mal L plus, falls einmal die Entscheidung »kaufen« gefällt wurde, K.

6-6

Mögliche Online-Algorithmen für das Ski-Leihen-Problem.

Strategie A (»reicher Macker«)
Kaufe gleich am ersten Tag.

Strategie B (»kurz probieren«)
Leihe die ersten 5 Tage und kaufe am 6. Tag, wenn bis dahin die Saison nicht zu Ende ist.

Strategie C (»länger probieren«)
Leihe die ersten 19 Tage und kaufe am 20. Tag, wenn bis dahin die Saison nicht zu Ende ist.

Strategie D (»sehr lange probieren«)
Leihe die ersten 39 Tage und kaufe am 40. Tag, wenn bis dahin die Saison nicht zu Ende ist.

Strategie E (»armer Studie«)
Leihe immer.

6.1.2 Klassifikation: Kompetitive Rate

Wie gut sind die Strategien?

6-7

Offenbar haben die Strategien alle Vor- und Nachteile. Bei der Strategie des reichen Mackers kann es sein, dass er 200 Euro ausgibt, obwohl die Saison nur einen Tag hat und 10 Euro (für einmal Leihen) ausgereicht hätte. Er gibt also *zwanzig Mal mehr aus als nötig*. Bei der Strategie des armen Studies kann es sein, dass er beliebig viel ausgibt (sehr lange Saison), obwohl 200 Euro ausgereicht hätten.

✎ Zur Übung

Bestimmen Sie für jede der Strategien, »wie schlimm es werden kann«: Um wie viele Mal höher können die Ausgaben gemäß dieser Strategien sein als wirklich nötig?
Für die Strategie »reicher Macker« ist der Faktor also 20, für die Strategie »armer Studie« sogar ∞.

Die kompetitive Rate.

6-8

Das Verhältnis von »Kosten gemäß der Online-Strategie« versus »Kosten, wenn man die Ereignisse schon vorher alle kennen würde« nennt man *kompetitive Rate*. Der Name rührt daher, dass man die Online-Strategie gegen die beste Offline-Strategie »antreten lässt«.

▶ Definition: Kompetitive Rate

Gegeben seien ein Online-Problem und ein Online-Algorithmus hierfür. Die *kompetitive Rate des Algorithmus* ist das Maximum über alle Eingaben von

$$\frac{\text{Maß der vom Online-Algorithmus gefundenen Lösung}}{\text{Maß der optimalen, von einem Offline-Algorithmus gefundenen Lösung}}$$

oder der Kehrwert (bei Maximierungsproblemen).
Die *kompetitive Rate des Online-Problems* ist die kompetitive Rate des besten Online-Algorithmus für das Problem.

Die kompetitive Rate des Ski-Leihen-Problems.

6-9

Alle möglichen Online-Strategien für das Ski-Leihen-Problem sind von der Form: »Leihe die ersten x Tage und kaufe am Tag $x+1$.« Die kompetitive Rate einer solchen Strategie ist

$$\frac{xL+K}{\min\{(x+1)L,K\}} = \max\left\{\frac{xL+K}{xL+L}, \frac{xL+K}{K}\right\},$$

denn der »Worst-Case« tritt immer dann ein, wenn gerade am Tag $x+1$ der letzte Tag der Saison ist: Hier hat die Online-Strategie $xL+K$ ausgegeben, ideal wäre es aber gewesen, entweder $x+1$ Mal zu leihen oder gleich am Anfang zu kaufen – je nachdem, was billiger gewesen wäre.
Die kompetitive Rate nimmt ihr Minimum ein, wenn die Brüche gleich sind, also für $x = K/L-1$. Hier ist die kompetitive Rate gerade $2-L/K$.

▶ Satz

Die kompetitive Rate des Ski-Leihen-Problems ist $2-L/K$.

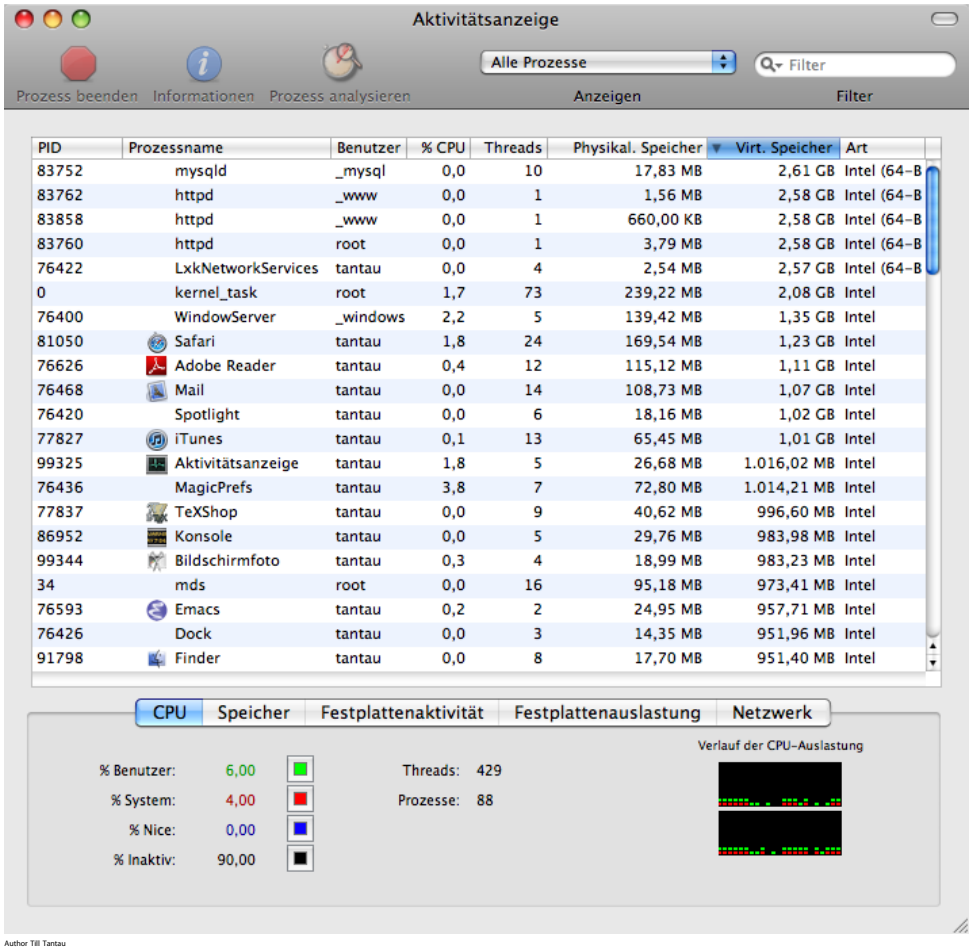
Mit anderen Worten:

- Die Strategie C (19 Tage leihen, dann kaufen) hat die beste kompetitive Rate.
- Allgemein hat die Strategie die beste kompetitive Rate, bei der man solange leiht, bis man so viel Geld ausgegeben hätte, als hätte man gleich gekauft. Dann sollte man aber kaufen.
- Dadurch gibt man höchstens doppelt so viel aus wie nötig.

6.2 Caching

6-11

Caching: Virtueller versus realer Speicher.



6-12

Caching lässt sich als Online-Problem darstellen

Das formale Caching-Problem

Gegeben sei ein Cache der Größe k.

Ereignisse »$request(x)$« für Seitennummern x

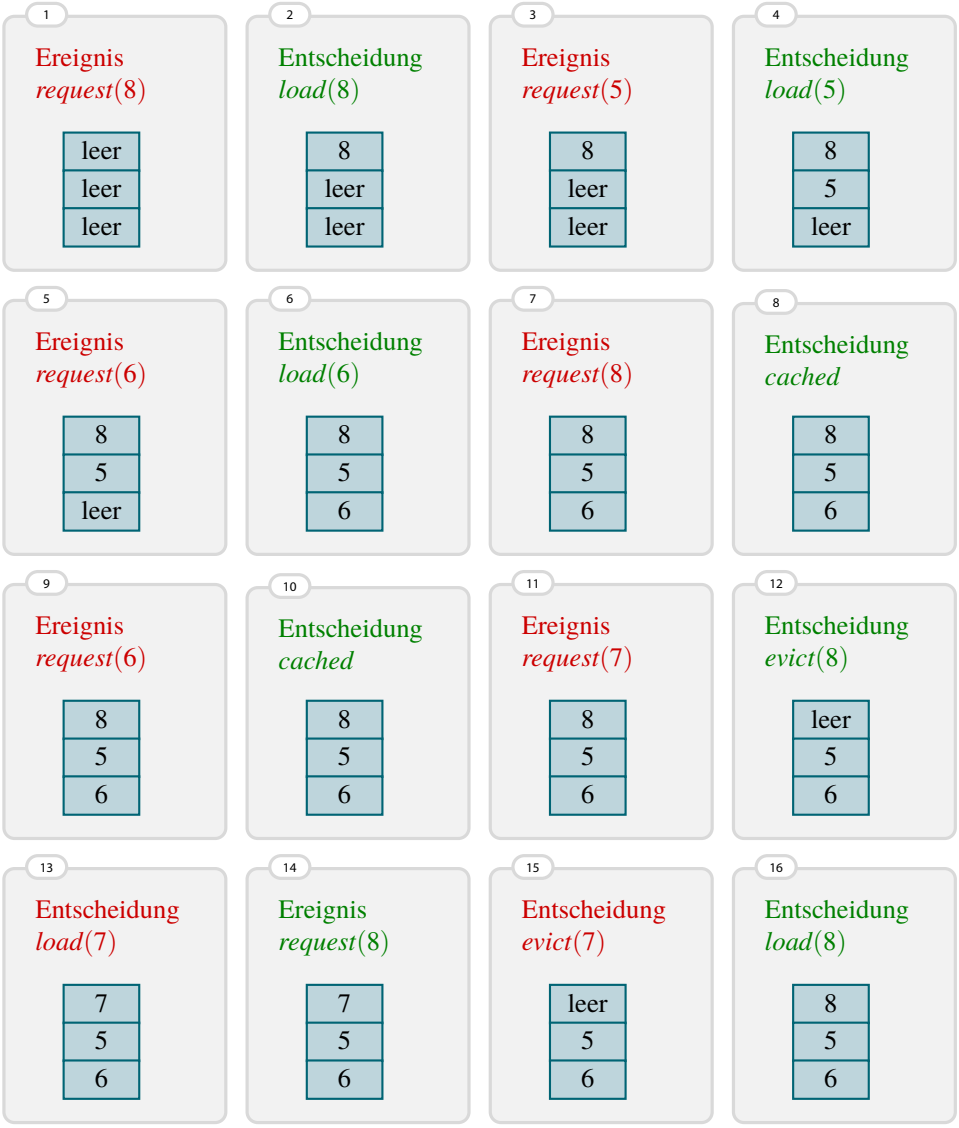
Entscheidungen Entweder »$cached$« oder »$evict(y)$« und/oder »$load(x)$«.

Bedingungen Zu jedem Schritt i enthält der Cache C_i eine Menge von Seitennummern. Hierfür gilt:

1. Zu Anfang enthält der Cache keine Elemente.
2. Falls die Entscheidung $cached$ ist, so muss $x \in C_i$ gelten und $C_{i+1} = C_i$.
3. Falls die Entscheidung $load(x)$ ist, so ist $C_{i+1} = C_i \cup \{x\}$.
4. Falls die Entscheidung $evict(y)$ ist, so ist $C_{i+1} = C_i \setminus \{y\}$.
5. Für alle Schritte gilt $|C_i| \leq k$.

Maß Anzahl der Load-Entscheidungen.

Ein Beispiel des Ablaufs eines Caching-Algorithmus. 6-13



6.2.1 Offline-Strategien

Ein Offline-Algorithmus für das Caching: Evict-Furthest-In-Future. 6-14

Nehmen wir zunächst an, *dass wir die Zukunft kennen*. Wenn nun der Cache voll ist, welchen Eintrag sollte man ersetzen? *Sicher* gilt: Wird ein Cache-Eintrag *nie wieder verlangt*, so kann er ersetzt werden. *Sicherlich* gilt: Einträge, die »lange Zeit« nicht mehr gebraucht werden, sollten eher ersetzt werden als solche, die »bald gebraucht werden«.

```
1 algorithm evict-furthest-in-future(request(x), C)
2   if x ∈ C then
3     return »cached«
4   else if |C| < k then
5     return »load(x)«
6   else
7     bestimme ein y ∈ C, so dass der Zeitpunkt des nächsten request(y) maximal ist
8     return »evict(y), load(x)«
```

6-15

Besser geht's nicht.

▶ Satz

Der Algorithmus Evict-Furthest-In-Future ist ein optimaler Offline-Algorithmus für das Caching-Problem.

Kommentare zum Beweis

Beweis. Man beweist diesen Satz mittels eines *Exchange-Arguments*. Dies funktioniert wie folgt:

1. Statt der vom Algorithmus erzeugten Liste von Entscheidungen betrachten wir eine *beliebige andere*. Intuitiv wird diese Liste »nicht ganz optimal« sein.
2. Wir betrachten dann, was passiert, wenn man diese Liste *verändert*, indem man »eine nicht optimale Stelle« gegen »eine optimale« austauscht (daher der Name »Exchange-Argument).
3. Entscheidend ist, dass das Maß der modifizierten Liste *gleich oder besser* als das Maß der ursprünglichen Liste ist.

Wiederholt man das Argument oft genug, so erhält man im Ergebnis die vom Evict-Furthest-In-Future-Algorithmus erzeugt Liste, *deren Maß folglich nicht schlechter ist als das einer beliebigen anderen Liste*.
Das schwierigere und entscheidende Argument ist, dass Evict-Entscheidungen immer so wie vom Algorithmus gefällt werden sollten. Dies sieht man so:
Nehmen wir an, für ein $evict(y)$ wurde $y \in C_i$ nicht nach der Regel des Algorithmus gewählt. Es gibt also ein $y' \in C_i$, so dass der nächste $request(y)$ früher kommt als $request(y')$.[1]
Betrachten wir nun die neue Folge, die entsteht, wenn man die Entscheidungsfolge wie folgt modifiziert:[2]

1. Die Entscheidung $evict(y)$ wird ersetzt durch $evict(y')$.
2. Die nächste $load(y)$ Entscheidung wird ersetzt durch $load(y')$.

Dann gilt:

1. Zwischen den beiden Entscheidungen wird nicht auf y' zugegriffen.
2. Nach der (neuen) $load(y')$ Entscheidung ist der Cache-Inhalt derselbe wie ursprünglich.

Wir fassen zusammen: Wiederholt man die obigen Argumente so lange wie möglich, so erhält man eine Folge, in der alle $evict(y)$ immer dasjenige y auswählen, das am längsten nicht mehr zugegriffen werden wird, und das Maß der entstandenen Folge ist nicht größer als das Maß der Ursprungsfolge.
Da man auch leicht sieht, dass Load-Entscheidungen nur unmittelbar nach einem zugehörigen Request gefällt werden brauchen und Evict-Entscheidungen nur gefällt werden brauchen, wenn der Cache voll ist, folgt die Behauptung. □

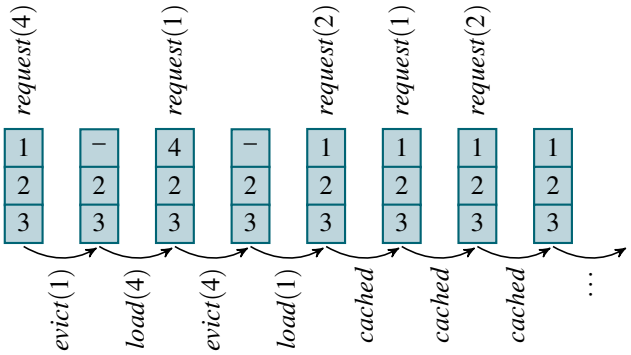
[1] Hier ist die Folge »nicht optimal«.

[2] Der »Exchange«-Schritt.

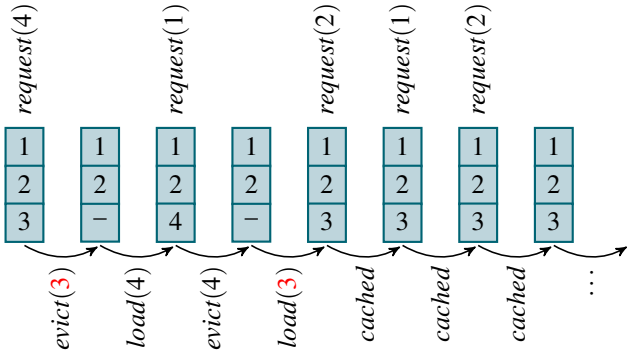
6-16

Das Exchange-Argument an einem Beispiel.

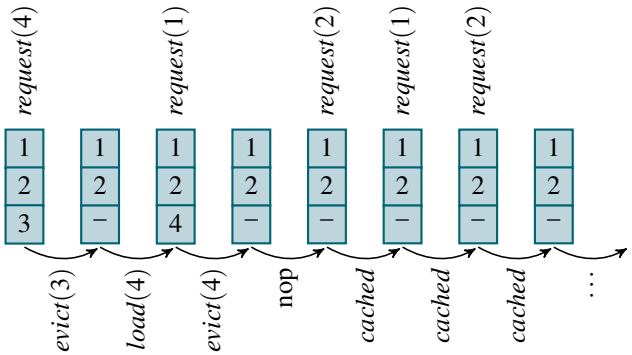
Eine Originalliste



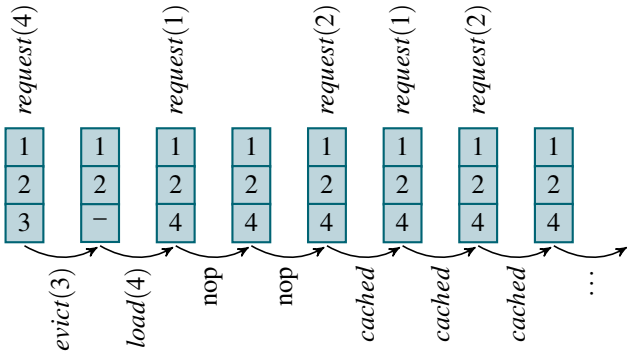
Die erste modifizierte Liste: Exchange-Argument für 1 und 3



Die zweite modifizierte Liste: $load(3)$ so spät wie möglich.



Die dritte modifizierte Liste: $evict(4)$ überflüssig



6.2.2 Online-Strategien

Online-Strategien: Wenn man die Zukunft nicht kennt. . .

6-17

»In der Wirklichkeit« ist die Zukunft natürlich nicht bekannt (siehe allerdings die Literaturhinweise). Ein Online-Algorithmus für das Caching-Problem muss folglich eine »Prognose« durchführen, welche Elemente in der Zukunft selten benötigt werden:

Evict-Least-Recently-Used-Strategie

Immer, wenn der Cache voll ist, entscheide $evict(y)$ für dasjenige y, für das am längsten kein Request-Ereignis vorlag.

✎ Zur Diskussion

Die »Hoffnung« bei der ELRU-Strategie ist, dass schon lange nicht mehr genutzte y auch in Zukunft nicht bald benötigt werden.
Wie sieht eine Folge von Request-Ereignissen aus, bei der diese Strategie besonders schlecht arbeitet?

6.2.3 Kompetitive Rate

6-18 Die kompetitive Rate von ELRU.

▶ Satz
Die Evict-Least-Recently-Used-Strategie ist k-kompetitiv (das heißt, ihrer kompetitive Rate ist höchstens k) bei einer Cache-Größe von k.

Beweis. Man betrachte eine beliebige Eingabe von Ereignissen und nun k aufeinander folgende Evict-Entscheidungen des ELRU-Algorithmus. Dann gilt:

1. Zwischen der ersten und letzten dieser Evict-Entscheidungen müssen Request-Ereignisse für k unterschiedliche Element vorgekommen sein.
2. Folglich muss jede andere Strategie hier auch mindestens eine Evict-Entscheidung fällen. □

6-19 Es geht nicht besser als mit ELRU.

▶ Satz
Jeder Online-Algorithmus für das Cache-Probleme hat eine kompetitive Rate von mindestens k. Das Problem selbst hat also die kompetitive Rate k.

Beweis. Dies zeigt man mit einem *Adversary-Argument*: Ein *Adversary* (Gegenspieler) betrachtet, welche Entscheidungen ein gegebener Online-Algorithmus bei einer Eingabefolge fällt. Nach jeder Entscheidung $evict(y)$ präsentiert der Adversary gerade $request(y)$ als nächstes Ereignis, was den Algorithmus zwingt, in jedem Schritt eine Evict-Entscheidung zu fällen. Der Evict-Furthest-In-Future-Algorithmus braucht hingegen nur jedes k-te Mal eine Evict-Entscheidung fällen (denn $k-1$ aufeinanderfolgende Request-Ereignisse beziehen sich immer auf Elemente, die schon im Cache sind). □

6.3 Scheduling

6-20 Worum es beim Scheduling geht.

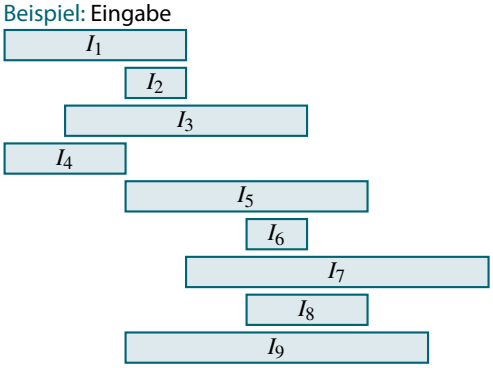
Author Turelio, Creative Commons Attribution Sharealike Licence

Beim *Scheduling* geht es darum, die zeitliche Abfolge von *Jobs* festzulegen, so dass *Bedingungen* eingehalten werden.

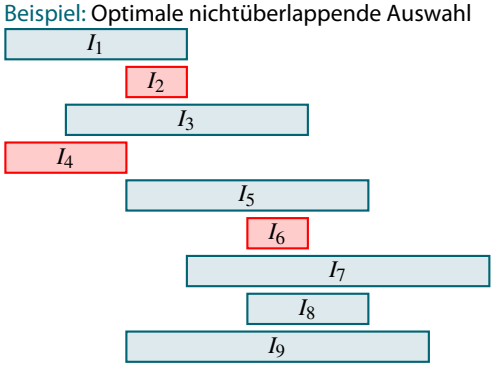
6-21 Das einfachste Scheduling-Problem.

▶ Definition: Intervall-Scheduling
Gegeben seien Intervalle $I_j = [s_j, f_j)$ mit $j \in \{1,\dots,n\}$, wobei s_j die *Startzeit* und f_j die *Endzeit* bezeichne. Ziel ist es, eine *möglichst große Auswahl* $J \subseteq \{1,\dots,n\}$ zu treffen, so dass für je zwei $j, j' \in J$ gilt $I_j \cap I_{j'} = \emptyset$.

Beispiel: Eingabe



Beispiel: Optimale nichtüberlappende Auswahl



6.3.1 Offline-Strategien

Das Scheduling-Problem kann mit einem Greedy-Verfahren gelöst werden. 6-22

Zur Erinnerung: *Greedy-Verfahren* finden Lösungen, indem sie *iterativ* jeweils *Teillösungen* schrittweise erweitern und dabei *»gierig«* die Erweiterung *lokal optimal* durchführen. Greedy-Verfahren finden *nicht immer* optimale Lösungen.

Greedy-Algorithmus für das Intervall-Scheduling

```
1 algorithm earliest-finishing-first(I_1, ..., I_n)
2   J ← ∅
3   X ← {1, ..., n}
4   while X ≠ ∅ do
5     j ← dasjenige j ∈ X für das f_j minimal unter allen j ∈ X ist
6     J ← J ∪ {j}
7     X ← X \ {k | I_k ∩ I_j ≠ ∅}
8   return J
```

Greedy führt zum Erfolg. 6-23

▶ Satz

Der Earliest-Finishing-First-Algorithmus findet immer ein optimales Schedule.

Zum Beweis siehe Übungsaufgabe 6.1.

6.3.2 Online-Strategien

Die Online-Fassung des Problems. 6-24

▶ Definition: Online-Fassung des Intervall-Scheduling-Problems

Ereignisse Intervalle $I_j = [s_j, f_j)$

Entscheidungen »schedule« oder »do not schedule«.

Bedingungen Alle Intervalle, für die »schedule« entschieden wurden, dürfen sich nicht überlappen.

Maß Anzahl der »schedule«-Entscheidungen.

Ein Greedy-Online-Algorithmus

Bei Eingabe I_j entscheide »schedule«, falls I_j sich mit keinem schon gewählten Intervall überlappt, sonst »do not schedule«.

6.3.3 Kompetitive Rate

6-25

Das Online-Problem ist »nicht kompetitiv«.

▶ Satz

Das Online-Problem Interval-Scheduling hat eine unendliche kompetititve Rate, das heißt, kein Online-Algorithmus für dieses Problem hat eine kompetititve Rate von höchstens k für irgendein k.

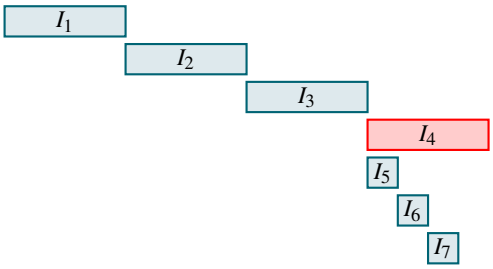
Kommentare zum Beweis

Beweis. Sei k fest und ein beliebiger Online-Algorithmus für das Interval-Scheduling-Problem gegeben. Wir betrachten folgenden Adversary:[1]

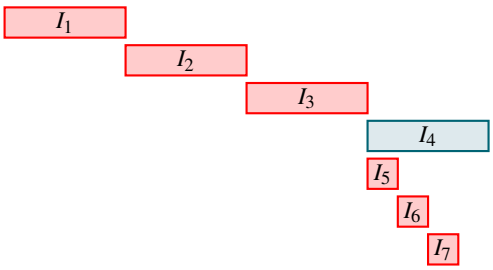
1. Er produziert die Intervalle $[1,2)$, $[2,3)$, $[3,4)$ und so weiter als Ereignisse, bis der Algorithmus zum ersten Mal »schedule« entscheidet.
2. Danach produziert er Intervalle der Größe $1/(k+1)$, die alle nebeneinander innerhalb dieses letzten Intervalls liegen.

[1] Hinweis, dass ein Adversary-Argument genutzt wird

Nach obiger Regel produziert der Adversary mindestens $k+2$ Intervalle.
Hier ein Beispiel dafür, welche Intervalle der Adversary produzieren könnte und welche der Algorithmus auswählen würde.



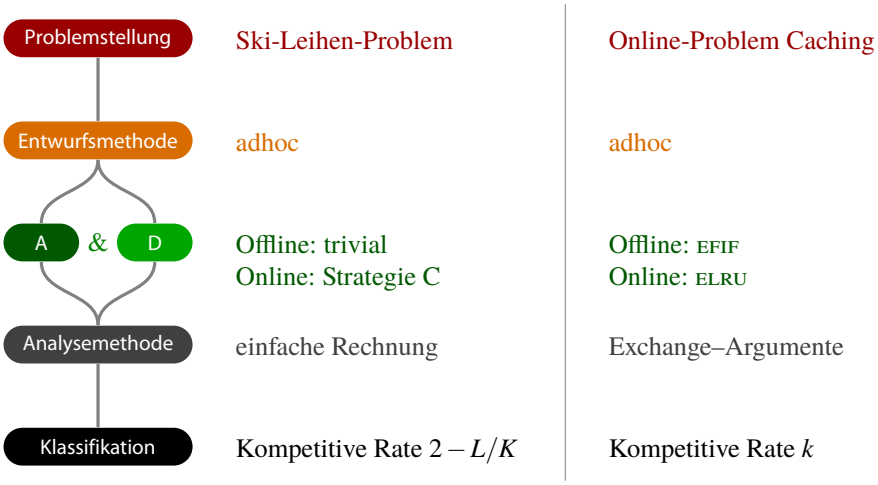
Optimal wäre es hingegen, gerade alle anderen Intervalle auszuwählen:

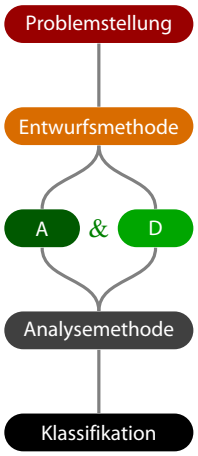


Wie man an den Beispielen (und auch leicht allgemein) sieht, wählt der Algorithmus nur ein Intervall aus, obwohl man $k+1$ Intervalle auswählen könnte. Folglich ist die kompetititve Rate schlechter als k. □

Zusammenfassung dieses Kapitels

6-26





Interval-Scheduling

Greedy

Offline: EFF
Offline: trivial

Adversay-Argument

Kompetitive Rate ∞
(nicht kompetitiv)

▶ Online- versus Offline-Algorithmen

Gegeben sei ein Online-Problem. Ein *Online-Algorithmus* für das Problem erhält die Ereignisse sequentiell und muss nach jedem Ereignis eine Entscheidung fällen. Ein *Offline-Algorithmus* für das Problem erhält alle Ereignisse auf einmal als Eingabe und muss dann eine Folge von Entscheidungen berechnen.

▶ Kompetitive Rate

1. Die *kompetitive Rate* eines Online-Algorithmus ist das maximale Verhältnis des Maßes der von ihm gefundenen Lösungen zum Maß der vom besten Offline-Algorithmus gefundenen Lösung.
2. Die *kompetitive Rate* eines Problems ist die minimale kompetitive Rate aller Online-Algorithmen für das Problem.

▶ Kompetitive Rate des Ski-Leihen-Problems

Die kompetitive Rate des Ski-Leihen-Problems ist höchstens 2.
Die Strategie hierzu: Leihe, bis man den Preis eines Paars ausgegeben hat, danach kaufe sofort.

▶ Kompetitive Rate des Caching-Problems

Die kompetitive Rate des Caching-Problems für Caches der Größe k ist k.
Ein optimaler *Offline*-Algorithmus ist Evict-Furthest-In-Future.

▶ Kompetitive Rate des Interval-Scheduling-Problems

Die kompetitive Rate des Interval-Scheduling-Problems ist unendlich.
Ein optimaler *Offline*-Algorithmus ist Earliest-Finishing-First.

Zum Weiterlesen

[1] Daryl J. Bem, Feeling the Future: Experimental Evidence for Anomalous Retroactive Influence on Cognition and Affect, *Journal of Personality and Social Psychology,* 100, 407–425, 2011.

Dies ist der in der Einleitung angesprochene Artikel, der sich mit der Frage beschäftigt, ob Menschen in die Zukunft sehen können. Der Artikel hat einen gewissen Aufruhr in der wissenschaftlichen Gemeinschaft ausgelöst; es empfiehlt sich sehr, die verschiedenen Repliken zu lesen.

Übungen zu diesem Kapitel

Übung 6.1 Earliest-Finishing-First ist optimal, mittel

Zeigen Sie, dass Algorithmus 6-22, der Earliest-Finishing-First-Algorithmus, immer eine Lösung mit minimalem Maß berechnet.

Tipp: Benutzen Sie ein Exchange-Argument. Zeigen Sie, dass es »nie falsch ist«, immer den Job auszuwählen, dessen Ende am frühesten ist. Gehen Sie dazu ähnlich wie bei Satz 6-15 vor: Betrachten Sie eine beliebige andere Auswahl und betrachten Sie die erste Stelle, wo die andere Auswahl von der des Earliest-Finishing-First-Algorithmus unterscheidet. Argumentieren Sie, dass man hier einen Austausch vornehmen kann, ohne das Maß zu verschlechtern.

Übung 6.2 k-Server-Problem auf den natürlichen Zahlen, mittel

Beim k-Server-Problem auf den natürlichen Zahlen ist eine Folge von *Anfragen* $a_1, \ldots, a_n$ mit $a_j \in \mathbb{N}$ gegeben. Ausgehend von einer ebenfalls gegebenen *Startkonfiguration* $c_0 = (c_1^0, \ldots, c_k^0) \in \mathbb{N}^k$ ist eine Folge von n *Konfigurationen* $c_i = (c_1^i, \ldots, c_k^i) \in \mathbb{N}^k$ gesucht, die für jede Anfrage a_j eine Konfiguration c_i mit $a_j \in \{c_1^i, \ldots, c_k^i\}$ enthält und die Summe $\sum_{i=1}^{n} \sum_{j=1}^{k} |c_j^{i-1} - c_j^i|$ minimert.

Das k-Server-Problem verallgemeinert viele Caching- und Scheduling-Probleme: Zum Beispiel könnten Anfragen für Hausnummern in einer Straße stehen, zu denen Briefe ausgestellt werden sollen. Die k Einträge der Startkonfiguration entsprechen Positionen von k Postboten, die diese Briefe mit möglichst wenig Aufwand austragen wollen. Gesucht ist eine Folge von Konfigurationen (Positionen der Postboten in der Straße), die alle Anfragen bedient (alle Briefe können ausgetragen werden) und dabei die Summe über den Differenzen benachbarter Konfigurationen (die Bewegungen der Postboten) minimiert.

1. Entwickeln Sie einen Offline-Algorithmus, der das k-Server-Problem optimal löst.
2. Formulieren Sie das k-Server-Problem als Online-Problem. Gehen Sie dabei wie in der Vorlesung vor und geben Sie die Ereignisse, Entscheidungen, Bedingungen und das Maß an.
3. Als ersten Versuch zur Lösung des Online-Problems betrachten wir folgende Greedy-Strategie: In Runde i bewege den Server mit geringstem Abstand zu a_i an die Position a_i. Geben sie ein Beispiel an, aus dem ersichtlich wird, dass dieser Algorithmus nicht k-kompetitiv für irgendein k ist.
4. Als zweiten Lösungsansatz betrachten wir die folgende Strategie, die unter dem Namen DOUBLE COVER bekannt ist: Zu jeder Runde i definieren wir $I_i = [\min_{j \in \{1,\ldots,k\}} c_j^{i-1}, \max_{j \in \{1,\ldots,k\}} c_j^{i-1}]$. Falls $a_i \notin I_i$, bewege den nächstgelegensten Server zu a_i. Falls $a_i \in I_i$, bewege die beiden nächstgelegensten Server (die nicht unbedingt gleich weit von a_i entfernt sein müssen) mit gleicher Geschwindigkeit Richtung a_i. Wenn einer der Server a_i erreicht, stoppe beide Server.
   Simulieren Sie den Algorithmus DOUBLE COVER für Ihr Beispiel aus Punkt 3. Wie ist die Rate zwischen der optimalen und der von DOUBLE COVER gefundenen Lösung?

7-1

Kapitel 7
Analysemethode: Amortisierung

Amortisierte Kosten sind reale Kosten plus Rückstellungen

Lernziele dieses Kapitels

1. Konzept der amortisierten Analyse verstehen
2. Die Potential-Methode einsetzen können

7-2

Inhalte dieses Kapitels

Worum es heute geht

Die amortisierte Analyse ist ein Ausflug der Theoretischen Informatik in die Finanzwelt. Allerdings bekommen die Informatiker die Begriffe wie immer nicht richtig hin: Wo eine Bankerin von »Rückstellungen« sprechen würde, spricht man in der Theorie von »Potentialen«, wo sie von »Preisen« sprechen würden, spricht man in der Theorie von »amortisierten Kosten«. Ignoriert man aber diese Unbeholfenheiten in der Ausdrucksweise, so lässt sich doch festhalten, dass die Theorie der amortisierten Kosten eine gute Idee aus der Finanzwelt umsetzt: *Wenn man heute schon genau weiß, welche Kosten in der Zukunft entstehen werden, dann macht es Sinn, jetzt schon Rückstellungen hierfür zu bilden.*

Wie der Name schon sagt, ist die amortisierte Analyse eine *Analysemethode* – sie hat weder auf Algorithmen und Datenstrukturen noch auf deren Entwurf einen Einfluss. Sie kommt immer dann zum Einsatz, wenn Algorithmen eine Folge von Operationen abarbeiten, deren Laufzeit stark unterschiedlich sein kann und bei denen diese Laufzeiten auch noch voneinander abhängen. Typisch ist es beispielsweise, dass spätere Operationen sehr teuer werden können, aber nur dann, wenn vorher viele billige Operationen durchgeführt wurden.

Hierzu ein Beispiel: Sie wohnen in einer Etagenwohnung und wollen alle paar Jahre mal renovieren. Eine solche »Renovieren-Operation« ist nur teuer, wenn vorher viele »Ein-Jahr-Wohnen-Operationen« durchgeführt wurden. Hingegen ist es recht billig zu renovieren, wenn dies kurz vorher bereits einmal geschehen ist. Aus diesem Grund sind für eine beliebige Folge von »Renovieren-Operationen« und »Ein-Jahr-Wohnen-Operationen« die *durchschnittlichen* Kosten einer Operation eher niedrig. Wenn man dies nun formal zeigen möchte, so verliert man schnell die Übersicht, wie teuer die einzelnen Operationen nun sind, denn hier hängt es ja von deren Reihenfolge ab (zugegeben, bei einem einfachen Beispiel wie diesem behält man wohl doch noch den Überblick; wenn es aber viele Operationen gibt, die in komplexer Weise voneinander abhängen, so ist man verloren). Die Idee der *Rückstellung* hilft nun weiter: In jedem Jahr bilden Sie eine kleine Rückstellung auf einem Konto. Das so angesparte Geld nutzen Sie dann, wenn eine Renovierung ansteht, so dass die Renovierung

selbst Sie idealerweise gar nichts kostet. Wenn man nun schaut, wie viel Kosten pro Jahr entstehen, so ist sofort klar, dass diese konstant sind: In normalen Jahren müssen Sie nur die Rückstellung bezahlen (ein kleiner, konstanter Betrag) und das Renovieren kostet Sie quasi »nichts«, denn sie wird aus den Rückstellungen bezahlt.

7.1 Einführung zur Amortisierung

7.1.1 Fallbeispiel I: Stacks

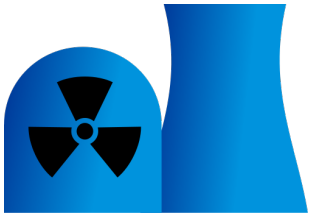
7-4

Von Atomkraftwerken. . .

Der Regierung von Dunkelland möchte die *Kernenergie* einführen. Sie kann:

1. Ein neues AKW *bauen* lassen.
2. Ein bestehendes AKW *abreißen* lassen.
3. Nach jedem GAU: auf einmal k *viele* AKW *abreißen* lassen.

Die Regierung beauftragt die Firma *Happy Atoms GmbH & Co. KG*.



Frage für heute

Wie viel wird der Auf- und Abbau der AKWs über die Jahre kosten?

7-5

. . . und Stacks

Auf einem *Stack* seien die folgenden Operationen erlaubt:

1. push(x)
2. pop
3. multipop(k): Es werden die obersten k Elemente des Stacks entfernt

Die Frage neu formuliert

Wie viel wird eine längere Folge von solcher Operationen »kosten«?

7.1.2 Fallbeispiel II: Zähler

7-6

Teure Zähler.

Normalerweise dauert das *Erhöhen einer Zahl* nur *einen Takt*. Heute soll es aber darum gehen, wie lange dies dauert, wenn man *die Zahl nur bitweise modifizieren kann.*

Speicherung von Zahlen in Arrays.

Zur Speicherung einer Zahl a benutzen wir

- einen Array A, wobei
- $A[i]$ gerade das i-te Bit von a ist (beginnend beim LSB und Zählung bei 0).

Beispielsweise gilt für $a = 110_2$, dass $A[0] = 0$, $A[1] = 1$ und $A[2] = 1$.

```
void increment (int[] A)
{
  int i = 0;
  while (A[i] == 1) {
    A[i] = 0;
    i++;
  }
  A[i] = 1;
}
```

7.1.3 Worst-Case-Kosten

Wie teuer ist eine Folge von Stackoperationen? 7-7

Gegeben sei eine *Folge der Länge n von Stackoperationen* wie push(a), push(b), push(a), multipop(2), push(c), ...

Fragestellung

Wie lange wird es dauern, diese Folge abzuarbeiten? Dabei nehmen wir an:

1. Ein Push dauert eine Zeiteinheit.
2. Ein Pop ebenfalls.
3. Ein multipop(k) dauert $\min\{s,k\}$ Zeiteinheiten, wobei s die aktuelle Stackhöhe ist, denn wir benötigen so viele Pop-Operationen.

Eine *einfache Worst-Case-Abschätzung* liefert:

▶ **Satz**

Eine Folge von n Stackoperationen dauert $O(n^2)$ lange.

Beweis. Jedes multipop(k) kann höchstens $n-1$ lange dauern. Da es n Operationen gibt, folgt die Laufzeit. □

Wie teuer ist eine Folge von Inkrements? 7-8

Gegeben sei eine *Folge von Inkrement-Aufrufen der Länge n.*

Fragestellung

Wie lange wird es dauern, diese Folge abzuarbeiten?

Wir sollen also abschätzen, wie lange folgendes Programm braucht:

```
int[] A = new int [64]; // Reicht, wenn n ein long ist

for (int j = 0; j < n; j++) {
  increment(A);
}
```

Eine *einfache Worst-Case-Abschätzung* liefert:

▶ **Satz**

Eine Folge von n Inkrement-Aufrufen dauert $O(n \log n)$ lange.

Beweis. Die Schleife in `increment` kann $\log n$ mal durchlaufen werden. Da es n Aufrufe gibt, folgt die Laufzeit. □

Zusammenfassung der Worst-Case-Kosten der Operationen. 7-9

Operation	Reale Worst-Case-Kosten
push(a)	1
pop	1
multipop(k)	$\min\{s,k\}$
increment	$\log_2 n$

7.1.4 Amortisierte Kosten I

7-10

Die erste Worst-Case-Abschätzung ist zu pessimistisch.

Betrachten wir statt der Kosten von Stackoperationen die Kosten von Dunkelland:

Operation	Reale Worst-Case-Kosten
AKW bauen	1 Mrd. Euro
AKW abreißen	1 Mrd. Euro
Kleiner Atomausstieg, k abreißen	k Mrd. Euro

Big Idea

Ein AKW kann man erst abreißen, wenn es gebaut wurde!

▶ **Satz**

Die Kosten einer Folge der Länge n von Bau-, Abreiß- und Atomausstieg-Operationen sind höchstens $2n$ Mrd. Euro.

Erster Beweis. In einer Folge können höchstens n Bau-Operationen vorkommen (Kosten n Mrd. Euro) und *auch höchstens so viele abgerissen* werden, was insgesamt Kosten von höchstens $2n$ verursacht. □

7-11

Noch mehr Pessimismus.

0	0	0	0	0
0	0	0	0	1
0	0	0	1	0
0	0	0	1	1
0	0	1	0	0
0	0	1	0	1
0	0	1	1	0
0	0	1	1	1
0	1	0	0	0
0	1	0	0	1
0	1	0	1	0
0	1	0	1	1
0	1	1	0	0
0	1	1	0	1
0	1	1	1	0
0	1	1	1	1
1	0	0	0	0
1	0	0	0	1

Betrachten wir die Kosten von Inkrement-Aufrufen *genauer*:

1. Nur bei *jeder zweiten* Zahl muss die Schleife überhaupt durchlaufen werden.
2. Nur bei *jeder vierten* Zahl muss die Schleife mehr als einmal durchlaufen werden.
3. Nur bei *jeder achten* Zahl muss die Schleife mehr als zweimal durchlaufen werden.

Allgemein ist nur bei jeder 2^i-ten Zahl ein zusätzlicher Schleifendurchlauf nötig.

7-12

Die echten Kosten von mehreren Inkrements.

▶ **Satz**

Die Laufzeit einer Folge von n Aufrufen der Inkrement-Funktion liegt in $O(n)$.

Beweis. Wir summieren die Anzahl der Änderungen von Bits »spaltenweise«. Die Laufzeit ist dann:

$$\sum_{i=0}^{\lceil \log_2 n \rceil} \left\lfloor \frac{n}{2^i} \right\rfloor \leq n \sum_{i=0}^{\lceil \log_2 n \rceil} \frac{1}{2^i} < 2n.$$

□

Warum die Worst-Case-Abschätzung zu pessimistisch war. 7-13

Stacks
Ein multipop(k) kann zwar *sehr teuer sein*, aber nur dann, wenn es *vorher viele billige Operationen* gab. *Im Schnitt* ist ein multipop(k) billig: Da eine Liste von Operationen der Länge n nur $2n$ Kosten verursacht, sind die *durchschnittlichen Kosten jeder Operation* gerade 2.

Zähler
Ein Inkrement kann zwar *teuer sein*, aber nur dann, wenn es *vorher viele billige Operationen* gab. *Im Schnitt* ist ein Inkrement billig: Da eine Liste von n Inkrements nur $2n$ Kosten verursacht, sind die *durchschnittlichen Kosten eines Inkrements* gerade 2.

Idee der amortisierten Kosten. 7-14

Operation	Reale Worst-Case-Kosten	Amortisierte Kosten
push(a)	1	2
pop	1	2
multipop(k)	$\min\{s,k\}$	2
increment	$\log_2 n$	2

▶ **Definition:** Amortisierte Kosten allgemein
Eine Zuordnung von Kosten zu Operationen nennt man *amortisierte Kosten*, wenn für jede Folge von Operationen gilt:

$$\text{reale Kosten} \leq \text{Summe der amortisierten Kosten.}$$

Bemerkungen:

- Die *realen Kosten* einer einzelnen Operationen können *beliebig höher* sein als ihre amortisierten.
- Auch wenn hier »durchschnittliche Kosten« betrachtet werden, hat dies *nichts mit Average-Case-Analysen zu tun*.

Merke: *Amortisierte Analysen sind Worst-Case-Analysen.*

7.2 Die Potential-Methode

7.2.1 Die Idee der Rückstellung

Rückstellungen für Kosten in der Zukunft 7-15
Die Regierung von Dunkelland ärgert es, dass es nach einem GAU immer so teuer wird, wenn sie viele AKWs auf einmal abreißen lassen will. Sie beschließt deshalb, folgendes Gesetz zu erlassen:

> Happy Atoms muss bei der Bank pro derzeit gebauten AKWs 1 Mrd. Euro Rückstellungen bilden.

Wie hoch müssen die Preise von Happy Atoms nun sein?
Nehmen wir an, es gibt gerade s Atomkraftwerke in Dunkelland.

Aufbau In der Rechnung von Happy Atoms an die Regierung von Dunkelland finden sich folgende Posten:
1. 1 Mrd. Euro Baukosten und
2. 1 Mrd. Euro für Rückstellungen (vorher waren nur s Mrd. Euro nötig, nun $s+1$ Mrd. Euro).

Der Preis beträgt also 2 Mrd. Euro.

Abriss Hier möchte Happy Atoms kein Geld haben, denn:
1. Der Abriss kostet zwar 1 Mrd. Euro, aber
2. dies kann gerade aus der Differenz der Rückstellungen beglichen werden (vorher s Mrd. Euro, nun $s-1$ Mrd. Euro).

Kleiner Atomausstieg Auch hier braucht Happy Atoms kein Geld:
1. Der Abriss kostet zwar k Mrd. Euro, aber
2. dies kann gerade aus der Differenz der Rückstellungen beglichen werden (vorher s Mrd. Euro, nun $s-k$ Mrd. Euro).

7-16 Wie man mit Rückstellungen argumentiert.

▶ Satz
Die Kosten einer Folge der Länge n von Bau-, Abreiß- und Atomausstieg-Operationen sind höchstens $2n$ Mrd. Euro.

Zweiter Beweis. Wir verlangen, dass, wenn es s AKWs gibt, die *Rückstellungen* gerade s Mrd. Euro betragen. Dann sind Preise, die Happy Atoms pro Operation verlangen muss:

$$\text{Preis} = \text{reale Kosten} + \text{neue Rückstellungen} - \text{alte Rückstellung}.$$

Die so entstehenden Preise sind:

Operation	Preis
AKW bauen	2 Mrd. Euro
AKW abreißen	0 Mrd. Euro
Kleiner Atomausstieg, k abreißen	0 Mrd. Euro

Nun gilt:

$$\begin{aligned}\text{Gesamtpreis} = \text{reale Gesamtkosten} &+ \\ &\text{Endrückstellungen} - \text{Anfangsrückstellung}\end{aligned}$$

Da die *Rückstellung nie negativ ist* und sie *am Anfang 0 Euro betrug*, folgt:

$$\text{Gesamtpreis} \geq \text{reale Gesamtkosten}.$$

Da der Gesamtpreis offenbar höchstens $2n$ Mrd. Euro beträgt, folgt die Behauptung. □

7.2.2 Amortisierte Kosten II

7-17 Die Namen für Rückstellungen und Preise in der Theoretischen Informatik.
In der Theoretischen Informatik spricht man

- von *Potentialen* statt von »Rückstellungen« und
- von *amortisierten Kosten* statt von »Preisen«.

▶ Definition: Potentialfunktion
Eine *Potentialfunktion* Φ ordnet Datenstrukturen nichtnegative reelle Zahlen zu.

▶ Definition: Amortisierte Kosten für Potentialfunktionen
Eine Operation möge c_i reale Kosten verursachen und die Datenstruktur d_i in d_{i+1} verwandeln. Dann sind die *amortisierten Kosten* der Operation

$$a_i = c_i + \Phi(d_{i+1}) - \Phi(d_i).$$

7-18 Der Hauptsatz über die amortisierten Kosten.

▶ Satz
Sei Φ eine Potentialfunktion und $p_1, p_2, \ldots, p_n$ eine Folge von Operationen. Die i-te Operation möge Kosten c_i verursachen und die Datenstruktur d_{i-1} in d_i umwandeln. Dann gilt:

$$\sum_{i=1}^{n} c_i = \Phi(d_0) - \Phi(d_n) + \sum_{i=1}^{n} a_i.$$

Falls also $\Phi(d_0) = 0$ und $\Phi(d_n) \geq 0$, so gilt

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} a_i.$$

Merke
Ist die Potentialfunktion nie negativ und anfangs Null, so sind die realen *Gesamt*kosten höchsten die amortisierten *Gesamt*kosten.

Eine amortisierte Analyse.

7-19

▶ Satz
Die Ausführung einer Folge der Länge n von push(x), pop *und* multipop(k) *Operationen dauert höchstens* $O(n)$.

Beweis. Wir definieren eine *Potentialfunktion* wie folgt:

$$\Phi(S) = \text{Höhe des Stacks } S.$$

Offenbar ist $\Phi(S)$ initial 0 und nie negativ. Dann sind die amortisierten Kosten jeder Operation gerade

Operation	amortisierte Kosten
push(x)	2
pop	0
multipop(k)	0

Die amortisierten Kosten von n Operationen sind folglich höchstens $2n$ und somit auch die realen Kosten. □

Eine zweite amortisierte Analyse.

7-20

▶ Satz
Die Laufzeit einer Folge von n Aufrufen der Inkrement-Funktion liegt in $O(n)$.

Beweis. Wir definieren eine Potentialfunktion für einen Array A wie folgt: $\Phi(A)$ *sei gerade die Anzahl der* 1*en in* A. Offenbar ist dies anfangs 0 und nie negativ.
Die amortisierten Kosten eines Inkrements sind:

$$a_i = \underbrace{e+1}_{\text{reale Kosten}} + \Phi(A_i) - \Phi(A_{i-1}),$$

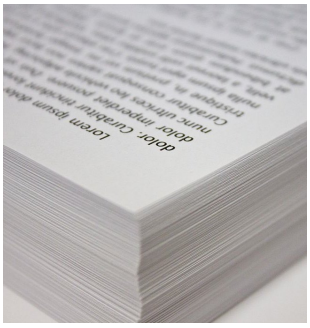
wobei e die Anzahl Einsen am Anfang des Arrays ist. Hier gilt immer $a_i = 2$, denn die Potentialänderung ist gerade $1-e$: Es werden e Einsen in Nullen verwandelt und eine Null in eine Eins.
Die amortisierten Kosten von n Operationen sind folglich höchstens $2n$ und somit auch die realen Kosten. □

7.3 Fallbeispiel III: Das Listen-Zugriffsproblem

Eine Online-Problemstellung.

7-21

Sie haben einen Papierstapel vor sich. Sie werden wiederholt gebeten, verschiedene Seiten darin zu suchen. Sie dürfen den Stapel nur von oben sequentiell durchsuchen. Haben Sie die Seite gefunden, können Sie diese beliebig viel weiter oben wieder einfügen.



Das formale Listen-Zugriffs-Problem

Ereignisse $request(x_i)$
Entscheidungen $moveto(d)$ (neue Tiefe von x_i).
Bedingungen $d \leq d_i$, wobei d_i die Tiefe von x_i im Stapel war
Maß $\sum_{i=1}^{n} d_i$

7.3.1 Online-Algorithmen

7-22

Eine gute Online-Strategie

Die Move-to-Front-Strategie

Entscheide immer $moveto(0)$.

✎ Zur Übung

Geben Sie eine Folge von Ereignissen an, für die diese Strategie eine schlechte kompetitive Rate hat.

7.3.2 Kompetitive Rate

7-23

Amortisierte Analyse der kompetitiven Rate

▶ Satz

Die kompetitive Rate der Move-to-Front-Strategie ist höchstens 2.

Beweis. Wir zeigen, dass die amortisierten Kosten der Move-to-Front-Strategie höchstens doppelt so groß sind wie die der optimalen Strategie. Hieraus folgt nach dem Hauptsatz, dass dies auch für die realen Kosten der Move-to-Front-Strategie gilt.

- Sei $x_1, \ldots, x_n$ eine feste Folge angefragter Elemente.
- Sei L_i jeweils die Liste nach dem i-ten Schritt gemäß den Entscheidungen eines optimalen Offline-Algorithmus.
- Sei L'_i jeweils die Liste nach dem i-ten Schritt gemäß den Entscheidungen der Move-to-Front-Strategie.
- Sei d_i die Tiefe von x_i in L_i und d'_i entsprechend für L'_i.
- Zwei Elemente der Liste bilden eine *Inversion*, wenn ihre Reihenfolge in L_i und L'_i unterschiedlich ist.

i	1	2	3	4	5	6	7	Gesamtkosten
x_i	c	b	a	c	b	a	c	
L_i	$\begin{matrix}a\\b\\c\end{matrix}$	$\begin{matrix}a\\b\\c\end{matrix}$	$\begin{matrix}a\\b\\c\end{matrix}$	$\begin{matrix}a\\b\\c\end{matrix}$	$\begin{matrix}a\\b\\c\end{matrix}$	$\begin{matrix}a\\b\\c\end{matrix}$	$\begin{matrix}a\\b\\c\end{matrix}$	
d_i	2	1	0	2	1	0	2	8
L'_i	$\begin{matrix}a\\b\\c\end{matrix}$	$\begin{matrix}c\\a\\b\end{matrix}$	$\begin{matrix}b\\c\\a\end{matrix}$	$\begin{matrix}a\\b\\c\end{matrix}$	$\begin{matrix}c\\a\\b\end{matrix}$	$\begin{matrix}b\\c\\a\end{matrix}$	$\begin{matrix}a\\b\\c\end{matrix}$	
d'_i	2	2	2	2	2	2	2	14
Inversionen	0	2	2	0	2	2	0	

Definiere eine Potentialfunktion wie folgt:

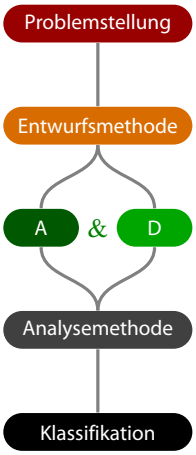
$$\Phi(L'_i) = \text{Anzahl Inversionen zu } L_i.$$

Offenbar ist Φ initial Null und nie negativ.
Die amortisierten Kosten jeder Entscheidung sind per Definition $a_i = d'_i + \Phi(L'_i) - \Phi(L'_{i-1})$.
Wir behaupten, dass gilt:

$$a_i \leq 2d_i.$$

Haben wir dies gezeigt, so sind wir fertig: Dann sind die amortisierten Gesamtkosten (und damit auch die realen Gesamtkosten) nämlich höchstens doppelt so wie die optimalen Kosten.
Zum Beweis von $a_i \leq 2d_i$ argumentieren wir so: Man stelle sich vor, x_i wird »langsam in L'_i nach oben geschoben«, wohingegen es »in L_i erstmal bleibt, wo es ist«. Dann wird, solange x_i unterhalb der Tiefe d_i ist, in jedem Schritt genau eine Inversion zu L_i aufgelöst, ab der Tiefe d_i hingegen wieder je eine neue Inversion geschaffen. Insgesamt verringert sich das Potential erst um $d'_i - d_i$, um sich dann um d_i zu erhöhen. Wird nun x_i auch in L_i langsam angehoben, so werden weitere Inversionen aufgelöst, das Potential verringert sich also weiter. Insgesamt ergibt sich eine maximale Potentialsteigerung von $d_i - (d'_i - d_i) = 2d_i - d'_i$. Also gilt $a_i \leq d'_i + 2d_i - d'_i = 2d_i$. □

Zusammenfassung dieses Kapitels



Stack-Verwaltung Zähler-Verwaltung	Listen-Zugriffs-Problem
adhoc	adhoc
Stack Zähler	Move-To-Front
Amortisierte Analyse	Amortisierte Analyse / Potential-Methode
lineare Kosten	2-kompetitiv

Amortisierte Kosten

Eine Zuordnung von Kosten zu Operationen nennt man *amortisierte Kosten*, wenn für jede Folge von Operationen gilt:

$$\text{reale Kosten} \leq \text{Summe der amortisierten Kosten.}$$

Potentialmethode

- Die *Potentialmethode* definiert amortisierte Kosten *implizit* durch eine *Potentialfunktion* Φ.
- Diese ordnet Datenstrukturen d_i nichtnegative reelle Zahlen zu.
- Die amortisierten Kosten einer Operation sind dann

$$a_i = \underbrace{c_i}_{\text{reale Kosten}} + \underbrace{\Phi(d_{i+1}) - \Phi(d_i)}_{\text{Potentialänderung}}.$$

Hauptsatz über amortisierte Kosten

Es gilt

$$\underbrace{\sum_{i=1}^{n} c_i}_{\text{reale Gesamtkosten}} = \underbrace{\Phi(d_0) - \Phi(d_n)}_{\text{Gesamtpotentialdifferenz}} + \underbrace{\sum_{i=1}^{n} a_i}_{\text{amortisierte Gesamtkosten}}.$$

Falls das Potential anfangs Null war und nie negativ, so gilt

$$\text{reale Gesamtkosten} \leq \text{amortisierte Gesamtkosten.}$$

(Für einzelne Operationen gilt dies hingegen gerade oft nicht.)

Zum Weiterlesen

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms,* zweite Auflage, MIT Press, 2001, Kapitel »Amortized Analysis«.

Übungen zu diesem Kapitel

Übung 7.1 Kompetitive Rate des DOUBLE COVER Algorithmus für das k-Server-Problem analysieren, schwer

In Übung 6.2 haben Sie das k-Server-Problem auf den natürlichen Zahlen und den DOUBLE-COVER-Algorithmus zu dessen Lösung kennengelernt. Im Folgenden soll mithilfe der Potential-Methode gezeigt werden, dass der DOUBLE COVER-Algorithmus für das k-Server-Problem k-kompetitiv ist. Hierzu verwenden wir folgende Potentialfunktion: Sei OPTIMAL ein optimaler offline Algorithmus für das k-Server-Problem und $q_0, q_1, \ldots, q_n$ eine Folge von Anfragen. Sei $c_i = (c_1^i, \ldots, c_k^i)$ die Konfiguration von OPTIMAL und $d_i = (d_1^i, \ldots, d_k^i)$ die Konfiguration von DOUBLE COVER in Runde i. Sei m_i der minimale Wert von $\sum_{j=1}^{k} |c_j^i - d_{f(j)}^i|$ über alle bijektiven Zuordnungen $f\colon \{1,\ldots,k\} \to \{1,\ldots,k\}$. Sei weiterhin $g_i = \sum_{(j,\ell)\in\{1,\ldots,k\}^2} |d_j^i - d_\ell^i|$. Wir betrachten die Potentialfunktion $\Phi(d_i) = k \cdot m_i + g_i$,

1. Geben Sie eine kurze Beispielsequenz von Anfragen an und berechnen Sie dazu eine optimale Lösung, die Lösung von DOUBLE COVER und die Werte für m_i, g_i, und $\Phi(d_i)$ in jeder Runde i.
2. Sei A ein beliebiger Offline-Algorithmus für das k-Server Problem. Wandeln Sie A in einen Algorithmus A' um, bei dem (1) Server nur auf Anfragen stehen bleibt und (2) der maximal einen Server zur gleichen Zeit bewegt. Folgern Sie hieraus, dass man sich bei der Betrachtung von optimalen Algorithmen auf Solche beschränken kann, die dieses Verhalten haben. Passen Sie Ihre optimale Berechnung aus 1. entsprechend an.
3. Zeigen Sie, dass die amortisierten Kosten a_i im Schritt i bezüglich der Potentialfunktion Φ maximal $k \cdot \Gamma_i$ mit $\Gamma_i = \sum_{j=1}^{k} |c_j^{i-1} - c_j^i|$ beträgt. Definieren Sie hierzu $\Delta_i = \sum_{j=1}^{k} |d_j^{i-1} - d_j^i|$ und stellen Sie die Gleichung für die amortisierten Kosten a_i bezüglich Φ auf. Betrachten Sie nun die Bewegungen von OPTIMAL und DOUBLE COVER in Runde i getrennt:
   - 3.1 Beweisen Sie, dass sich durch die Bewegung von OPTIMAL die Differenz $m_i - m_{i-1}$ um maximal Γ_i erhöht.
   - 3.2 Behandeln Sie die Fälle $q_i \notin I_i$ und $q_i \in I_i$ einzeln und zeigen Sie jeweils, dass sich durch die Bewegung von DOUBLE COVER die Differenz $m_i - m_{i-1}$ um mindestens Δ_i reduziert und die Differenz $g_i - g_{i-1}$ um maximal $(k-1)\cdot\Delta_i$ erhöht.

   Folgern Sie $a_i \leq k \cdot \Gamma_i$ und wenden Sie den Hauptsatz der amortisierten Analyse an, um hiermit zu zeigen, dass DOUBLE COVER k-kompetitiv.

Übung 7.2 Warteschlangen mit Kellern implementieren, mittel

Warteschlangen sind Datenstrukturen mit den Methoden enqueue(e) und dequeue$()$. Durch enqueue(e) reiht man ein Element am Ende der Warteschlange ein, mit dequeue$()$ entnimmt man das erste Element der Warteschlange.

Beschreiben Sie die Implementierung einer Warteschlange mithilfe von zwei Kellern und zeigen Sie mit der Potentialmethode, dass die amortisierten Kosten eines Methodenaufrufs $O(1)$ betragen.

Zur Erinnerung: Ein Keller ist eine Datenstruktur mit den Operationen push(e) und pop$()$, mit denen man ein Element in den Keller legen beziehungsweise das oberste Element aus dem Keller entnehmen kann. Wir betrachten Keller, bei denen die Laufzeit der Operationen in $O(1)$ liegt.

Übung 7.3 Binärer Zähler mit Rücksetzbefehl, mittel

Auf Seite 7-6 habe Sie die Methode

```
void increment (int[] A)
{
  int i = 0;
  while (A[i] == 1) {
    A[i] = 0;
    i++;
  }
  A[i] = 1;
}
```

zur Inkrementierung eines binären Zählers kennengelernt. Es wurde bereits mittels einer amortisierten Analyse gezeigt, dass sich die Laufzeit von n Aufrufen durch eine Funktion in $O(n)$ abschätzen lässt.

Neben der Inkrementierung betrachten wir nun als zusätzliche Operation die Zurücksetzung, bei der alle Bits der Zahl auf 0 gesetzt werden. Wir nehmen an, dass man eine Zeiteinheit zum Zurücksetzen eines Bits investieren muss.

1. Passen Sie den Pseudocode der Inkrementiermethode so an, dass sich der Algorithmus immer die Position des höchstwertigen 1-Bits merkt.
2. Geben Sie den Pseudocode einer Rücksetzmethode an, die ausgehend vom höchstwertigen 1-Bit alle Positionen bis zur Position 0 abläuft und 1-Bits auf 0 setzt.
3. Verwenden Sie die Potentialmethode, um zu zeigen, dass die Laufzeit für n Inkrementier- und Rücksetzoperationen $O(n)$ beträgt.

Übung 7.4 Warteschlangen mit Mehrfachentnahme, mittel

Erweitern Sie die Implementierung der Warteschlange aus Übung 7.2 um multi-dequeue(k), eine Operation, welche die ersten k Elemente aus der Warteschlange entnimmt. Wenden Sie die folgende Potentialmethode an, um zu zeigen, dass die amortisierten Kosten für jeden Aufruf von enqueue(e), dequeue() und multi-dequeue(k) konstant sind:

$$\Phi = 2 \cdot \text{Anzahl der Element auf dem Enqueue-Stack} + \text{Anzahl der Elemente auf dem Dequeue-Stack.}$$

Übung 7.5 Binärer Zähler mit Division, mittel

Der binäre Zähler aus der Vorlesung soll nun um eine Methode division-by-two() erweitert werden, welche die im Array gespeicherte natürliche Zahl durch 2 dividiert. Für den Zähler bedeutet dies, dass das LSB gelöscht und alle anderen Einträge um eine Position in dessen Richtung verschoben werden.

Geben Sie wie in Übung 7.3 eine Methode für die neue Operation an und zeigen Sie mithilfe der folgenden Potentialfunktion, dass die Laufzeit für n Inkrementier- und Divisionsoperationen $O(n)$ beträgt:

$$\Phi = \text{Anzahl der 1-Bits} + \text{Wert der Zahl.}$$

8-1

Kapitel 8
A&D: Union-Find
Vereinigt Euch!

8-2

Lernziele dieses Kapitels

1. Den optimalen Algorithmus zur Verwaltung disjunkter Mengen erklären können.
2. Die amortisierte Analyse des Algorithmus verstehen.

Inhalte dieses Kapitels

Worum es heute geht

In der etwas fortgeschrittenen Algorithmik stößt man oft auf Algorithmen und Datenstrukturen, deren Konstruktion kompliziert ist und die aufwändig zu verwalten sind – wer schon einmal einen AVL-Baum oder einen Fibbonacci-Heap implementieren durfte (höchstwahrscheinlich wohl aber eher »musste«), weiß, wovon ich spreche. Die Analyse solcher Strukturen ist dann in der Regel auch kein Zuckerschlecken, schließlich wollen viele Fälle und Probleme bedacht werden.

In seltenen Fällen liefert die Algorithmik uns aber Datenstrukturen oder Algorithmen, die zwar ganz einfach zu erklären sind, aber ziemlich schwierig zu analysieren sind. Ein Beispiel solch einer Datenstruktur sind die Union-Wälder, um die es in diesem Kapitel geht. Sie zu erklären ist wirklich einfach: Es handelt es sich um wurzelgerichtete Wälder, die drei Operationen unterstützen: (1) Einen neuen einzelnen Knoten zum Wald als eigenen Baum hinzufügen, (2) zwei Bäume vereinigen, indem die Wurzel des kleineren Baumes ein weiteres Kind der Wurzel des anderen wird und (3) von einem Knoten aus die Wurzel eines Baumes suchen und dabei alle Knoten auf dem Pfad zur Wurzel zu direkten Kindern der Wurzel zu machen. Ok, vielleicht doch keine »ganz« einfache Datenstruktur, aber kompliziert ist sie nun wirklich nicht.

Ganz anders stehen die Dinge bei der Analyse dieser Datenstruktur. Ältere Lehrbücher trauen sich den einfachen (!) Teil der Analyse nur in Abschnitten mit einem dicken Sternchen zu präsentieren, der wohl andeuten soll, dass die Formel-Orgien in den Analysen definitiv nicht jugendfrei sind. In den letzten Jahren wurden die Argumente in verschiedenen Veröffentlichungen zwar deutlich vereinfacht, wirklich einfach sind sie trotzdem noch nicht geworden. Insofern möchte ich auch für die in diesem Kapitel vorgestellte Analyse, die verschiedene Ideen aus der Literatur kombiniert, gerne eine FSK 16 Einstufung aussprechen – um einen mathematischen Hardcore-Porno, vor dem man empfindliche Studierendenseelen schützen müsste, handelt es sich aber nicht (wenn Sie an so etwas interessiert sind, schauen Sie doch mal in die Orginalliteratur).

8.1 Die Verwaltung disjunkter Menge

8.1.1 Die Problemstellung

Motivation: Der Algorithmus von Kruskal

8-4

```
1  procedure minimum-spanning-tree(vertices V, edges E, edge weights w)
2    sort the edges by w (if necessary)
3    S ← ∅
4    for each edge e = {u,v} in order of increasing weight do
5      if u and v are in different components of the graph (V,S) then
6        S ← S ∪ {{u,v}}
7    return minimum spanning tree (V,S)
```

Die Geschwindigkeit dieses Algorithmus hängt an zwei Fragen:

1. Wie schnell lassen sich die Kanten sortieren?
2. Wie schnell lässt sich die Frage beantworten, ob u und v in der gleichen Zusammenhangskomponente liegen?

Wir suchen eine Datenstruktur, um die *zweite* Frage schnell zu beantworten.

Gesucht: Eine Datenstruktur zur Verwaltung disjunkter Mengen.

8-5

Ziele

Wir suchen ein Datenstruktur zur *Verwaltung von Familien disjunkter Mengen* – in unserem Fall bilden jeweils die Knoten einer Zusammenhangskomponente eine Menge. Wir müssen Mengen *vereinigen können* und *testen können, ob Element in derselben Menge liegen.*

Ideen

- Jede Menge wird durch eines seiner Element *repräsentiert*, es heißt der *Repräsentant.*
- Es gibt eine Methode find(x), die den Repräsentanten der Menge *findet,* die x enthält.
- Zwei Elemente x und y sind genau dann in derselben Menge, wenn find$(x) =$ find(y).

Wir suchen also eine Datenstruktur, die folgende Operationen unterstützt:

init(x)	Erzeugt eine einelementige Menge mit dem Element x.
rep(x)	Gibt den Repräsentanten der Menge zurück, die x enthält.
unite(x,y)	Vereinigt die Mengen, die x und y enthalten.

Der Algorithmus von Kruskal mit unserer Datenstruktur

8-6

```
1  procedure minimum-spanning-tree(vertices V, edges E, edge weights w)
2    sort the edges by w (if necessary)
3    for each v ∈ V do
4      init(v)
5    S ← ∅
6    for each edge e = {u,v} in order of increasing weight do
7      f_u ← rep(u)
8      f_v ← rep(v)
9      if f_u ≠ f_v then
10       unite(f_u, f_v)
11       S ← S ∪ {{u,v}}
12   return minimum spanning tree (V,S)
```

8.1.2 Die Datenstruktur

8-7

Zur Diskussion

Machen Sie Vorschläge, wie Sie eine Datenstruktur für disjunkte Menge implementieren könnten!

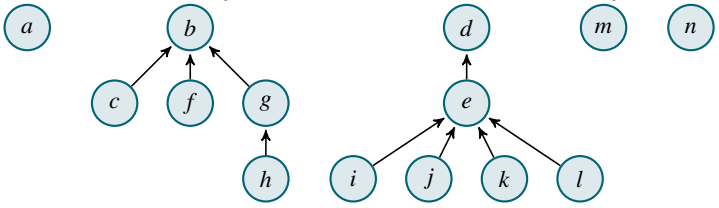
8-8

Der Union-Wald: Mengen als Bäume.

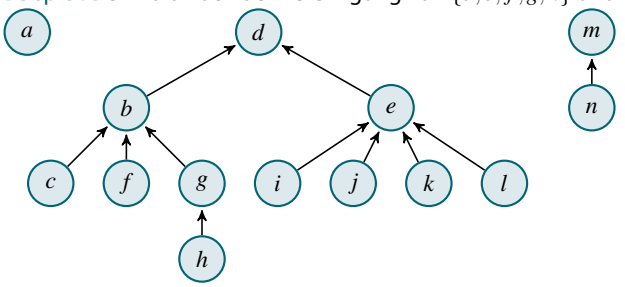
Struktur eines Union-Waldes

1. Die Elemente jeder Menge bilden einen (flachen) *Baum*, die Familie von Mengen damit einen *Wald*.
2. Der Repräsentant eines Baumes ist *seine Wurzel*.
3. Wir *vereinigen* zwei Mengen (= Bäume), indem wir die Wurzel eines Baumes zu einem weiteren Kind der Wurzel des anderen Baums machen.

Beispiel: Ein Wald für $\{\{a\},\{b,c,f,g,h\},\{d,e,i,j,k,l\},\{m,n\}\}$



Beispiel: Der Wald nach der Vereinigung von $\{b,c,f,g,h\}$ und $\{d,e,i,j,k,l\}$



8-9

Je flacher der Baum, desto besser.

Die find-Operation ist umso teurer, je tiefer der Baum ist. Deshalb gilt: Je flacher der Baum, desto besser.
Drei Ideen:

1. Jeder Knoten hat ein Attribut »*rank*«, das bei 1 beginnt und das wir nur »widerwillig erhöhen«.
2. Bilden wir die Vereinigung zweier Bäume, so hängen wir immer den Baum mit kleinerem Wurzelrang unter den mit größerem Wurzelrang. (Sind die Ränge gleich, muss erst »widerwillig« einer explizit erhöht werden.)
3. Wenn wir eine Wurzel suchen und sowieso schon alle Knoten auf dem Pfad zur Wurzel »anfassen«, können wir sie auch gleich auf dem Rückweg *zu Kindern der Wurzel machen.*

8-10

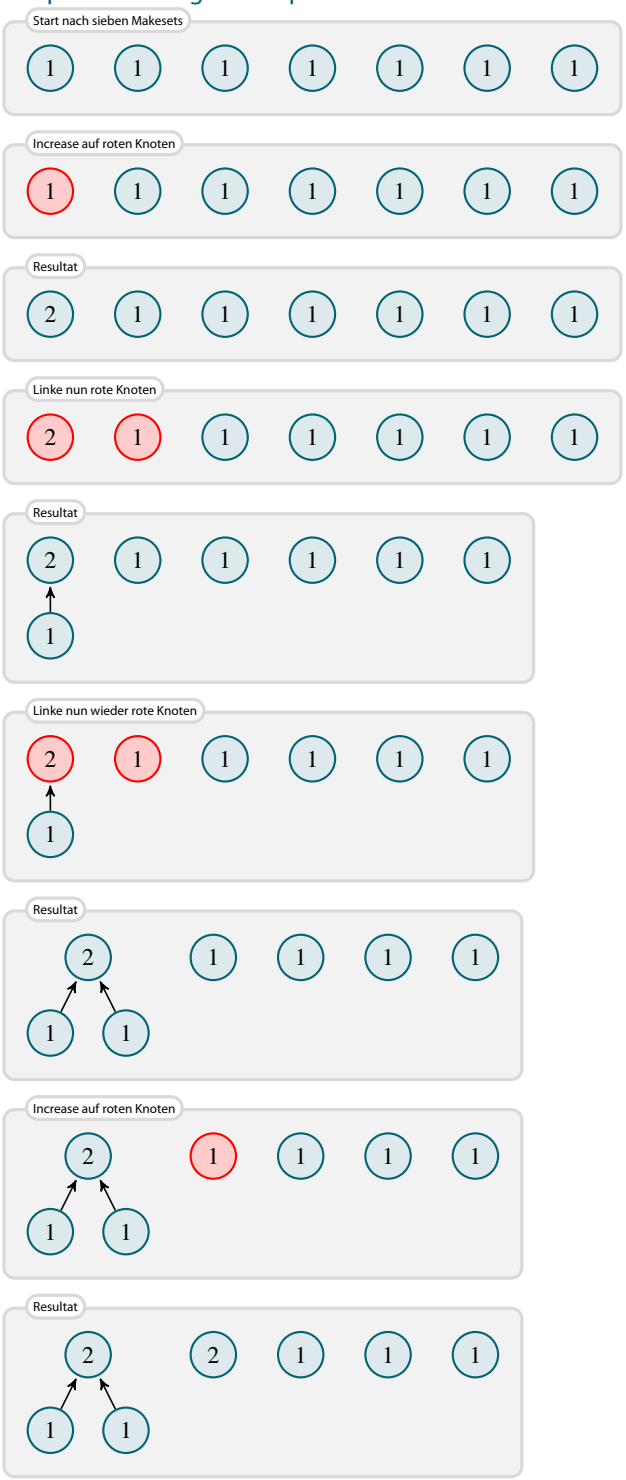
Vier Methoden, auf denen sich die Operationen »init«, »rep« und »unite« leicht aufbauen lassen.

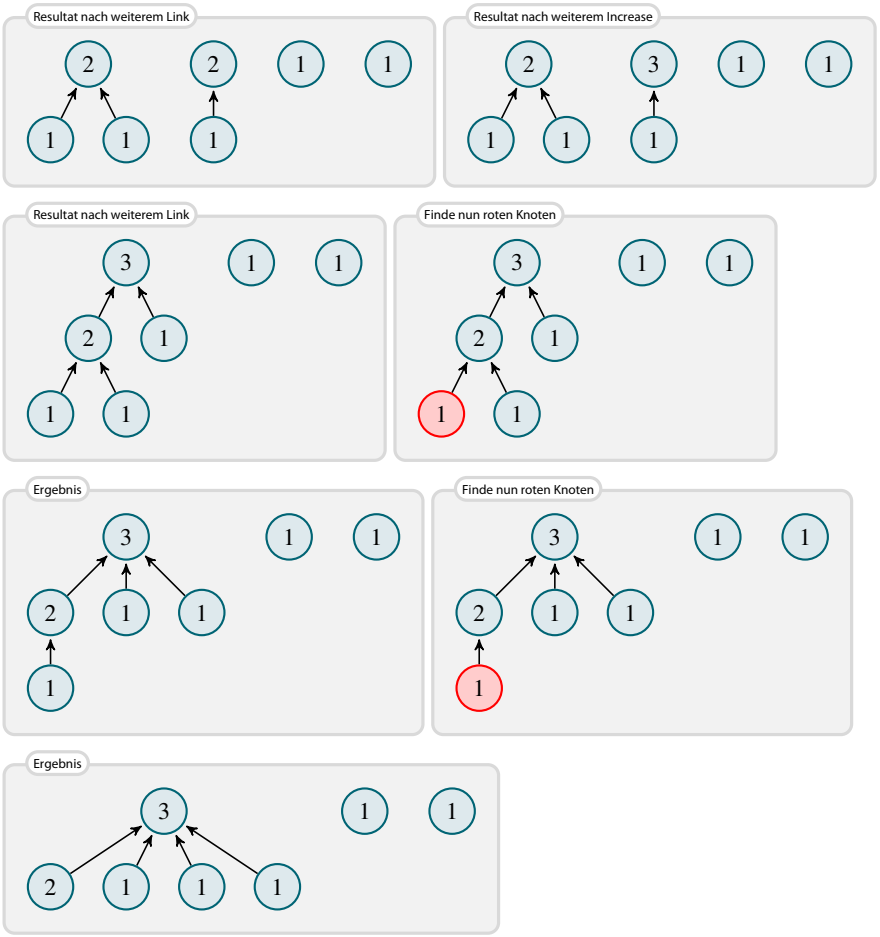
```
1  procedure makeset(x)
2    x.parent ← null
3    x.rank ← 1
4
5  procedure increase(x)
6    // Precondition: x is a root
7    x.rank ← x.rank +1
8
9  procedure link(x,y)
```

```
10    // Precondition: x and y are roots and have different ranks
11    case x.rank < y.rank: x.parent ← y
12    case x.rank > y.rank: y.parent ← x
13
14  procedure find(x)
15    if x.parent = null then
16      return x
17    else
18      x.parent ← find(x.parent)
19      return x.parent
```

Beispiel einer Folge von Operationen.

8-11





8-12

✎ Zur Übung

Bestimmen Sie den Wald, der aufgrund der folgenden Folge von 20 Operationen entsteht:

1. $makeset(a)$, $makeset(b)$, $makeset(c)$, $makeset(d)$, $makeset(e)$,
2. $increase(a)$, $increase(a)$, $increase(a)$, $increase(a)$,
3. $increase(b)$, $increase(b)$, $increase(b)$,
4. $increase(c)$, $increase(c)$,
5. $link(d,c)$,
6. $link(e,c)$,
7. $link(b,c)$,
8. $link(a,b)$,
9. $find(c)$,
10. $find(e)$.

8-13

Ein paar einfache Beobachtungen.

1. Der Elternknoten eines Knoten hat *immer einen echt höheren Rang*.
2. Ist ein Knoten erstmal keine Wurzel mehr, *ändert sich sein Rang nie wieder*.
3. Auf einem Pfad zur Wurzel steigen die Ränge strikt an.

8.2 Analyse

Wie lange dauert eine Folge von Operationen?

8-14

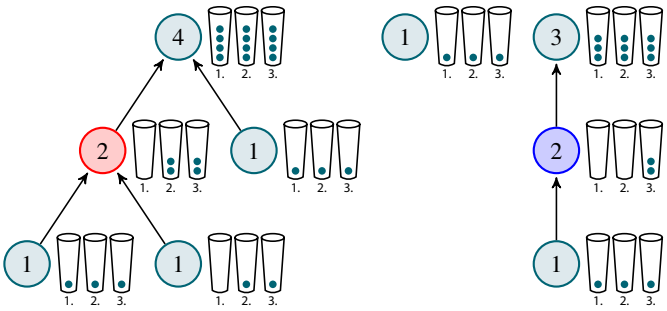
Wir suchen eine *obere Schranke für die Laufzeit* von n von Operationen (Makeset, Increase, Find, Link). Dazu führen wir eine amortisierte Analyse durch mit einer Potentialfunktion. Es wird *leicht* sein zu sehen, dass die amortisierten Kosten $O(n\alpha(n))$ sind, wenn wir die Funktion $\alpha(n)$ groß genug wählen. Es wird *schwieriger* sein zu sehen, dass wir als $\alpha(n)$ die *umgekehrte Ackermann-Funktion wählen können*.

8.2.1 Begriffe: Eimer und Münzen

Grundideen der Analyse

8-15

Wir stellen uns vor, dass neben jedem Knoten $\alpha(n)$ *Eimer stehen,* nummeriert von 1 bis $\alpha(n)$. (Wir werden $\alpha(n)$ erst später bestimmen.) In die Eimer werden wir nach und nach gleichmäßig *Münzen* einfüllen (bei Makeset- und Increase-Operationen). Bei einer Find-Operation werden wir *maximal eine Münze pro Knoten entfernen.* Tun wir dies, so entnehmen wir die Münze dem *ersten nichtleeren Eimer.* Seine Nummer nennen wir den *Level* des Knotens. Das *Potential* eines Waldes ist *die Summe aller Münzen in allen Eimern in allen Knoten.*



- Jeder Knoten hat $\alpha(n) = 3$ Eimer.
- Das *Potential* des roten Knotens ist 4, das des blauen 2, das Gesamtpotential ist 40.
- Der *Level* des roten Knotens ist 2, des blauen 3. Alle Wurzelknoten haben Level 1.

8.2.2 Regeln: Ein- und Auszahlung

Die genauen Regeln, wie Münzen in die Eimer kommen.

8-16

Regel für *makeset*(x)

Ein Knoten x beginnt mit einer Münze pro Eimer.

Regel für *increase*(x)

Füge jedem Eimer von x eine Münze hinzu.

Regel für *link*(x,y)

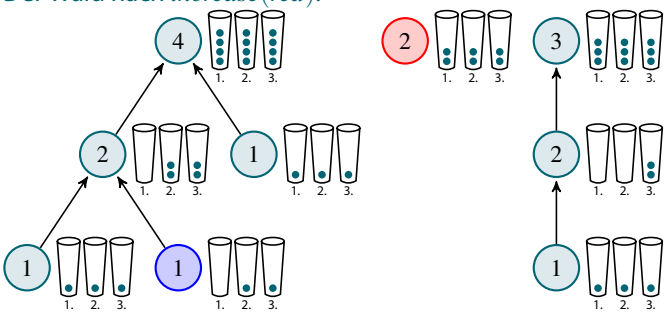
Tue nichts.

Regel für *find*(x)

Seien x_1 bis x_k die Knoten auf dem Pfad von x zur Wurzel w, mit $x_1 = x$ und $x_k = w$. Für $i \in \{1,\ldots,k-1\}$ entferne eine Münze bei x_i, wenn es ein $j \in \{i+1,\ldots,k-1\}$ gibt mit $x_i.level = x_j.level$. (»Pro Level wird beim wurzelnächsten Knoten keine Münze entfernt.«)

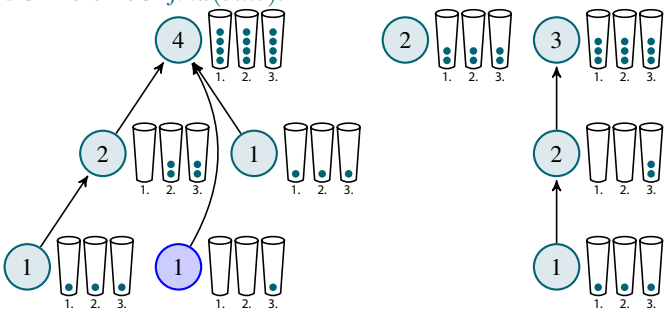
Der Wald nach *increase*(red).

8-17



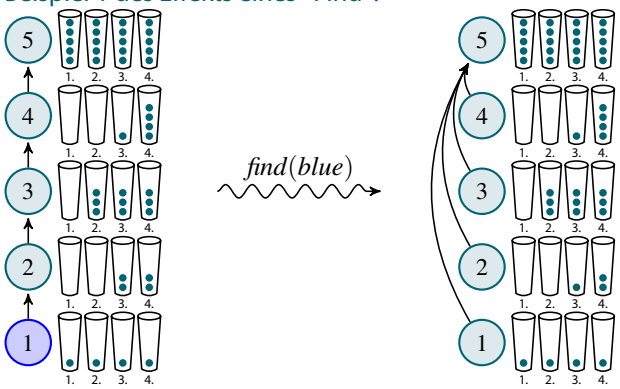
8-18

Der Wald nach $find(blue)$.



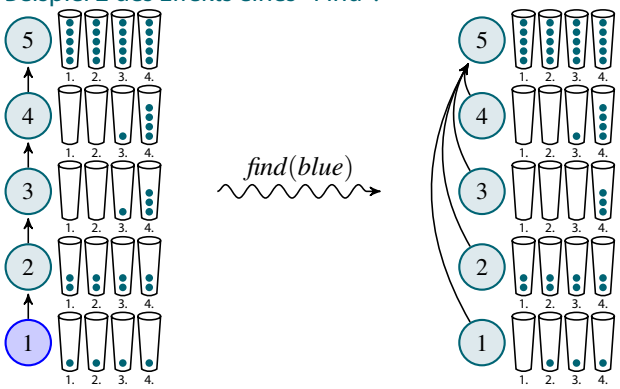
8-19

Beispiel 1 des Effekts eines »Find«.



8-20

Beispiel 2 des Effekts eines »Find«.



Skript

Für alle, die die obige Beschreibung mit Eimern und Münzen mal »mathematisch aufgeschrieben sehen möchten«, hier die Definitionen noch einmal etwas formaler:

▶ **Definition:** Potentialfunktion

Für eine Folge $o_1, \ldots, o_n$ der Makeset-, Increase-, Link- oder Find-Operationen sei D_n der resultierende Wald gemäß dem Algorithmus von Folie 8-10. Wir erweitern (konzeptionell) die Attribute der Knoten (derzeit »parent« und »rank«) um ein weiteres Attribut »coins«. Wie sich die Werte dieses Attributs verändern, wird später definiert. Es gilt aber:

$$\Phi(D_i) = \sum_{x \text{ ist Knoten in } D_i} x.coins.$$

▶ **Definition:** Level

Für einen Knoten x sei der *Level* des Knoten definiert als

$$x.level = (\alpha(n)+1) - \lceil x.coins/x.rank \rceil$$

▶ **Definition:** Änderung des Coins-Attributs

Durch eine Operation o_i möge sich der Wald D zu D' ändern. Die Coins-Attribute ändern sich dann in Abhängigkeit davon, wie o_i lautet, wie folgt:

makeset(x) In D' gilt $x'.coins = \alpha(n)$. (In jedem Eimer ist initial eine Münze, was bei $\alpha(n)$ Eimern gerade $\alpha(n)$ Münzen entspricht.)

increase(x) Es gilt $x'.coins = x.coins + \alpha(n)$. (Pro Eimer wird eine Münze hinzugefügt.)

link(x,y) Keine Änderungen der Coins-Attribute.

find(x) Seien x_1 bis x_k die Knoten auf dem Pfad von x zur Wurzel r, mit $x_1 = x$ und $x_k = r$. Für $i \in \{1,\dots,k-1\}$ sei

$$x_i'.coins = \begin{cases} x_i.coins - 1, & \text{wenn es ein } j \text{ mit } i < j < k \text{ und } x_i.level = x_j.level \text{ gibt,} \\ x_i.coins, & \text{sonst.} \end{cases}$$

✎ Zur Übung 8-21

Bestimmen Sie das Potential des Waldes, der aufgrund der folgenden Folge von 20 Operationen entsteht ($\alpha(n) = 3$):

1. *makeset*(a), *makeset*(b), *makeset*(c), *makeset*(d), *makeset*(e),
2. *increase*(a), *increase*(a), *increase*(a), *increase*(a),
3. *increase*(b), *increase*(b), *increase*(b),
4. *increase*(c), *increase*(c),
5. *link*(d,c),
6. *link*(e,c),
7. *link*(b,c),
8. *link*(a,b),
9. *find*(c),
10. *find*(e).

8.2.3 Abschätzung der amortisierten Kosten

Die amortisierten Kosten pro Operation sind höchstens $\alpha(n)$. 8-22

▶ Satz

Sei $\alpha(n)$ groß genug gewählt, dass niemals alle Eimer bei irgendeinem Knoten leer werden. Dann sind die amortisierten Kosten jeder Makeset-, Increase-, Link- und Find-Operation höchstens $O(\alpha(n))$ und damit die realen Gesamtkosten einer Folge n solcher Operationen höchstens $O(n \cdot \alpha(n))$.

Beweis. Die amortisierten Kosten einer Operation sind die realen Kosten plus die Änderung des Potentials. Bei uns ist die Potentialänderung gerade die Anzahl an hinzugefügten Münzen. Die realen Kosten von *makeset*(x) und *increase*(x) sind 1 und es werden $\alpha(n)$ Münzen hinzugefügt, was $O(1+\alpha(n))$ amortisierte Kosten ergibt. Die realen Kosten von *link*(x,y) sind 1 und es werden keine Münzen hinzugefügt, was $O(1)$ als amortisierte Kosten ergibt. Die realen Kosten von *find*(x) ist die Länge des Pfades von x zur Wurzel. Bis auf $\alpha(n)$ viele Knoten entfernen wir aber für jeden Knoten auf diesem Pfad eine Münze; das Potential wird also genau um die Pfadlänge gesenkt – bis auf die $\alpha(n)$ Knoten, die dann genau die amortisierten Kosten darstellen. □

8.2.4 Abschätzung des maximalen Levels

8-23

Wie schnell leeren sich die Eimer?

Unsere Analyse ergibt eine »um so bessere Abschätzung«, je kleiner wir $\alpha(n)$ wählen können. Wir müssen daher wissen, »wie schnell die Eimer leer werden«. Für einen Knoten x sei $p = x.parent$. Wir zeigen gleich, dass

1. der erste Eimer erst leer ist, wenn $p.rank > 2 \cdot x.rank$;
2. der zweite Eimer erst leer ist, wenn $p.rank > 2^{x.rank} = 2 \uparrow x.rank$;
3. der dritte Eimer erst leer ist, wenn $p.rank > 2 \uparrow\uparrow x.rank$,
4. der vierte Eimer erst leer ist, wenn $p.rank > 2 \uparrow\uparrow\uparrow x.rank$,
5. und so weiter.

Die »Pfeil-Notation« ist nach Knuth wie folgt definiert:

$$a \uparrow^n b = \begin{cases} a \cdot b & \text{für } n = 0, \\ a & \text{für } b = 1 \text{ und} \\ a \uparrow^{n-1} (a \uparrow^n (b-1)) & \text{sonst.} \end{cases}$$

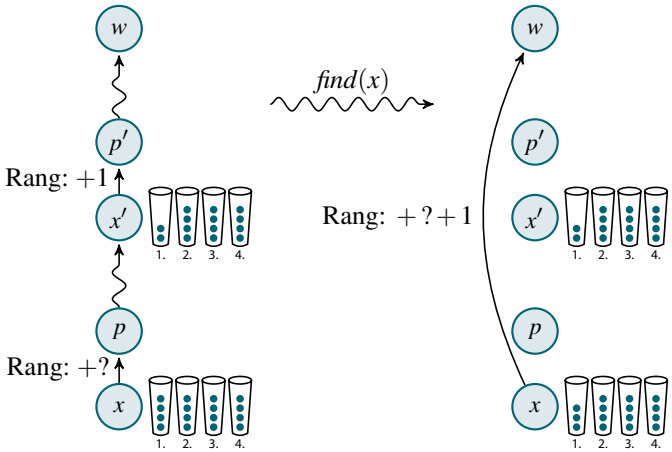
8-24

Level 1: Der erste Eimer

▶ Lemma

Sei x ein Knoten, dessen erster Eimer leer ist. Dann gilt $p.rank \geq 2 \cdot x.rank$.

Beweis. Wenn zum allerersten Mal eine Münze entfernt wird, gilt schon $x.rank < p.rank$. Es wird nur dann eine Münze entfernt, wenn später auf dem Pfad von x zur Wurzel ein weiterer Knoten x' kommt, dessen erster Eimer ebenfalls nicht leer ist. Da $x'.rank < p'.rank$ und x als neuen Elternknoten die Wurzel w des Pfades bekommt mit $p'.rank \leq w.rank$, folgt, dass der neue Elternknoten w von x nach dem »Umhängen« einen um *mindestens* 1 *höheren Rang* hat. Damit der erste Eimer von x komplett leer wird, muss sich also $x.rank$ mal der Rang des Elternknotens um mindestens 1 erhöht haben. □



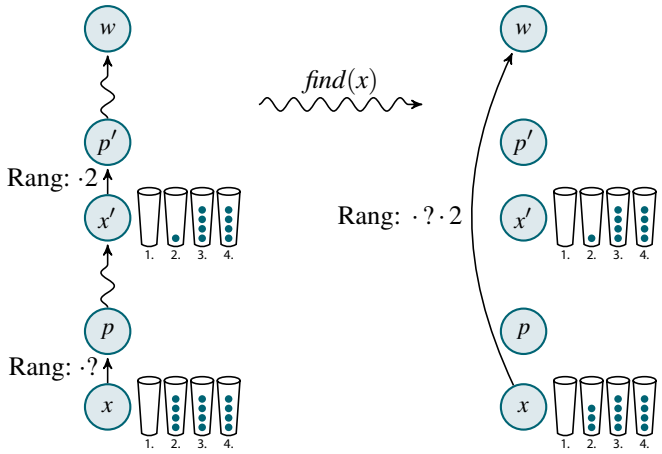
8-25

Level 2: Der zweite Eimer

▶ Lemma

Sei x ein Knoten, dessen zweiter Eimer leer ist. Dann gilt $p.rank > 2^{x.rank}$.

Beweis. Es wird nur dann eine Münze aus dem zweiten Eimer entfernt, wenn später auf dem Pfad von x zur Wurzel ein weiterer Knoten x' kommt, dessen zweiter Eimer ebenfalls nicht leer ist. Für diesen gilt nach dem vorigen Lemma $2x'.rank < p'.rank$ und somit auch $2x.rank < w.rank$. Da w neuer Elternknoten von x wird, *verdoppelt* sich der Rang des Elternknotens jedes Mal, wenn eine Münze aus dem zweiten Eimer entfernt wird. Damit der zweite Eimer von x komplett leer wird, muss sich also $x.rank$ mal der Rang des Elternknotens verdoppelt haben. □



Level 3: Der dritte Eimer

8-26

▶ Lemma

Sei x ein Knoten, dessen dritter Eimer leer ist. Dann gilt $p.rank > 2 \uparrow\uparrow x.rank$.

Beweis. Ähnlich wie in den vorigen zwei Lemmas gilt: Wir entnehmen nur dann eine Münze dem dritten Eimer, wenn es ein x' mit leerem zweiten Eimer später auf dem Pfad gibt. Nach dem vorigen Lemma gilt beim Sprund von x' zu p', dass der Rang von $x'.rank$ auf $p'.rank > 2^{x'.rank}$ springt – und damit dann auch der von $x.rank$ auf $w.rank$. Jede Entnahme einer Münze aus dem dritten Eimer erhöht somit den Rang des Elternknoten von r auf 2^r. Damit der Eimer leer wird, müssen wir also »$x.rank$ oft potenzieren«, was gerade $2 \uparrow\uparrow x.rank$ ergibt. □

Die weiteren Eimer und $\alpha(n)$.

8-27

▶ Lemma

Sei x ein Knoten, dessen i-ter Eimer leer ist. Dann gilt $p.rank > 2\underbrace{\uparrow \cdots \uparrow}_{i-1 \text{ Mal}} x.rank = 2 \uparrow^{i-1} x.rank$.

Beweis. Per Induktion über i; das Argument ist genau das aus den Lemmata vorher. □

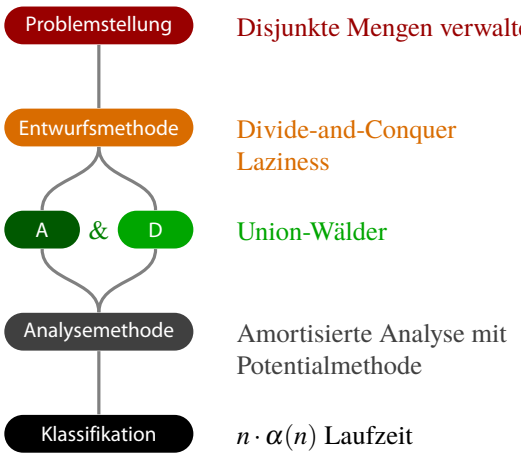
▶ Folgerung

Wählen wir $\alpha(n)$ als kleinestes i mit $n \leq 2 \uparrow^i 2$, so wird nie ein Eimer leer werden.

Dieses $\alpha(n)$ ist gerade die *umgekehrte Ackermann-Funktion*.

Zusammenfassung dieses Kapitels

8-28



▶ Satz

Eine beliebige Folge von n Makeset-, Increase-, Find- und Link-Operationen kann in Zeit $O\big(n \cdot \alpha(n)\big)$ abgearbeitet werden.

▶ Beweisidee

1. Das Potential ist die Summe aller Münzen in den $\alpha(n)$ Eimern pro Knoten.
2. Makeset und Increase platzieren eine Münze in jedem Eimer.
3. Für die Kosten eines Find bezahlen wir mit einer Münze pro Knoten, außer bei den jeweils ersten Knoten eines Levels, was nur $\alpha(n)$ viele sind.
4. Der i-te Eimer von x kann erst leer werden, wenn der Elternrang mindestens $2 \uparrow^{i-1} x.rank$ beträgt.
5. Ist daher $\alpha(n)$ die umgekehrte Ackermann-Funktion, so gehen uns nie die Münzen aus.

(Man kann zeigen, dass *jede* Datenstruktur für die Verwaltung disjunkter Mengen diese Kosten hat.)

Teil IV

Zufall als Entwurfsmethode

Es gibt im Deutschen sogar ein Sprichwort, dass uns dringend davon abrät, dem obigen Motto der nachfolgenden Kapitel zu folgen:

> »Überlasse nichts dem Zufall.«

Und es stimmt ja: Computer müssen sehr akribisch vorgehen, wenn sie Probleme lösen. Schon ein umgekipptes Bit in den Daten – geschweige denn im Code – bringt einen Rechner schnell zum Blue-Screen. Trotz dieser berechtigten Einwände ist »der Zufall« ein ausgesprochen nützlicher Geselle, wenn es um den Entwurf von Algorithmen geht.

Als allen Leserinnen und Lesern hoffentlich wohlvertrautes Beipsiel betrachten wir den Quicksort-Algorithmus mit der – zugegebenermaßen eher dümmlichen – Wahl des jeweils ersten Elements des zu sortierenden Arrays als Pivot-Element. Wenn nun die zu sortierenden Daten bereits sortiert sind, dann hat Quicksort eine sehr miese Laufzeit von $\Omega(n^2)$. Sind die Eingabedaten hingegen *zufällig,* so ist die Laufzeit im Erwartungswert nur $O(n \log n)$. Der Quicksort ist aber nur das bekannteste Beispiel eines Algorithmus, der mit zufälligen Daten sehr gut umgehen kann, mit »speziellen« Daten hingegen eher schlecht.

Vor diesem Hintergrund kann man bei einem Algorithmus wie Quicksort entweder einfach »hoffen«, dass die Daten halt schön zufällig sind und, wenn sie es nicht sind, eben in den sauren Laufzeitapfel beißen. Alternativ kann man aber eben auch *selber den Zufall ins Spiel bringen:* Bei Quicksort ist dies ganz einfach, indem man statt dem ersten Element einfach ein zufälliges Element als Pivot-Element nutzt.

Generell geht es in diesem Teil des Skripts um die »Geisteshaltung«, den Zufall in das Design von Algorithmen einzubauen. Wie wir sehen werden, bringt dies in vielen Fällen enorme Beschleunigungen in den Algorithmen – ohne dass die Korrektheit der Ergebnisse darunter leiden müsste.

9-1

Kapitel 9
A&D: Suchen in hashbaren Daten

Kuckucks-Hash-Tabellen sind perfekt, dynamisch und verdammt schnell

9-2

Lernziele dieses Kapitels

1. Ideen und Methodik des Hashings auffrischen
2. Hash-Tabellen mit veränderlicher Größe analysieren können

Inhalte dieses Kapitels

Worum es heute geht

Was ist eigentlich ein »Hash«? Wie es sich für ein kurzes englisches Wort gehört, hat es ziemlich viele Bedeutungen, denn generell kann man bei einem guten englischen Wörterbuch eine fast beliebige Kombination von vier Buchstaben suchen und mit ziemlicher Sicherheit kommen eine große Anzahl von Bedeutungen heraus – von denen außer den Wörterbuchschreibern noch niemand gewusst hat. Beispielsweise bezeichnet das doch eher selten gebrauchte Substantiv »balk« solche unterschiedliche Dinge wie »an illegal motion made by a baseball pitcher that may decieve a base runner«, »a roughly squared timber beam«, »any area on a pool or billiard table in which play is restricted in some way« und bekanntermaßen natürlich auch »a ridge left unplowed between furrows«.

Ein »Hash« ist nun laut Webster-Wörterbuch »a dish of cooked meat cut into small pieces and recooked, usually with potatoes«. Es gibt Hashes in verschiedenen Varianten, die auch liebevoll aufgezählt werden; die offenbar kulinarisch belesenen Autoren des Wörterbuches haben so auch an Vegetarier im Rahmen eines »hash of raw tomatoes, chilies, and coriander« gedacht. Entsprechend bedeutet dann »to hash«, »make meat or other food into a hash«, man kann »hashing« also grob als »verhackstückseln« übersetzen. (Anderseits bedeutet »to make a hash of something« so viel wie wie »es verpatzen«.)

Wie verhackstückselt man nun aber Daten? Dies erscheint doch als eher gefährlicher, vielleicht sogar verbotener Vorgang. Tatsächlich wird beim Hashing ein Objekt genommen und daraus durch wildes »Herumrechnen« eine einzelne Zahl errechnet – der so genannte *Hashwert* des Objekts. Ähnlich wie man beim gekochten Hash auch nicht mehr erkennen kann, von welchem Rindvieh es stammt, so kann man einem Hashwert auch nicht mehr ansehen, von welchem Objekt es stammt. Wichtig ist beim Hashing eigentlich nur, dass unterschiedliche Objekte (möglichst) unterschiedliche Hashwerte bekommen sollten.

In diesem Kapitel möchte ich zunächst Ihr Wissen über Hashtabellen »etwas auffrischen«, aber auch ein neues Konzept einführen: die Idee der dynamischen Größenanpassung. Bei den meisten Hashtabellen weiß man in dem Moment, wo man sie angelegt, noch nicht, wie große die Tabelle wird. Wählt man nun die Größe zu klein, so werden viele Hashverfahren extrem langsam oder funktionieren gar nicht mehr. Wählt man sie zu groß, so wird unter Umständen sehr viel Speicher verschwendet. Die dynamische Größenanpassung schafft hier

Abhilfe: Immer, wenn die Auslastung der Tabelle (der »Load-Factor«) zu sehr von einem Idealwert abweicht, wird die Größe der Tabelle angepasst. Dies bedeutet aber, dass die Tabelle komplett neu aufgebaut werden muss, was in der Regel ziemlich lange dauert. Hier setzt nun aber die uns mittlerweile wohlvertraute amortisierte Analyse ein: Wir werden zeigen, dass die amortisierten Kosten der Größenänderungen konstant sind. Mit anderen Worten: Wir bekommen die automatische Größenanpassung von Hashtabellen geschenkt.

Das Wort »Hash« ist übrigens im Vergleich zu den Alternativen, die der Thesaurus bereithält, eher langweilig. Es würde die Informatikliteratur sicherlich etwas auflockern, redete man statt von Hashwerten von *mishmash values*, die man statt in eine Hash-Tabelle in eine *gallimaufry table* einfügt.

9.1 Grundlagen zu Hash-Tabellen

Wie schnell geht es wirklich?

9-4

Was Suchbäume können

Zeitbedarf der Basis-Operationen bei 2-3-Suchbäumen, wenn n Elemente gespeichert sind:

Operation	Zeitbedarf
$insert(k,v)$	$O(\log n)$
$delete(k)$	$O(\log n)$
$find(k)$	$O(\log n)$

Der Platzbedarf ist ebenfalls $O(n)$.

Was wir gerne hätten

Idealer Zeitbedarf der Basis-Operationen, wenn n Elemente gespeichert sind:

Operation	?
$insert(k,v)$	$O(1)$
$delete(k)$	$O(1)$
$find(k)$	$O(1)$

Der Platzbedarf soll ebenfalls $O(n)$ betragen.

Das Versprechen der Hash-Tabellen

- Eine *Hash-Tabelle* versucht, die drei Basisoperationen in *konstanter Zeit* erledigt zu bekommen.
- Standard-Verfahren wie *lineares Sondieren* benötigen jedoch *im Worst-Case Zeit* $O(n)$.
- *Perfekte Hash-Tabellen* unterstützen hingegen zumindest das Suchen in *Worst-Case Zeit* $O(1)$.

Die Dinge, nach denen wir heute suchen

9-5

1. Jedes Objekt besteht aus einem *Schlüssel* und einem *Wert*.
2. Die *Schlüssel* sind Zahlen aus dem »Schlüsseluniversum« $U = \{0,1,2,\ldots,N-1\}$ (oder lassen sich als solche kodieren).

Beispiel

Konten in einem Graphen

Schlüssel Die Knotennummern

Werte Informationen, die an den Knoten »hängen«

Übersetzung der Anforderungen nach Java

```
interface HashMap.Entry <K extends Object, V>
{
  K    getKey   ();
  V    getValue ();
  void setValue (V value);
}
```

9-6

Welche Grundoperationen sind erlaubt?

Operationen, die wir heute unterstützen wollen

1. Einfügen von Elementen
2. Löschen von Elementen
3. Suchen nach Elementen anhand von Schlüsseln

Übersetzung der Anforderungen nach Java

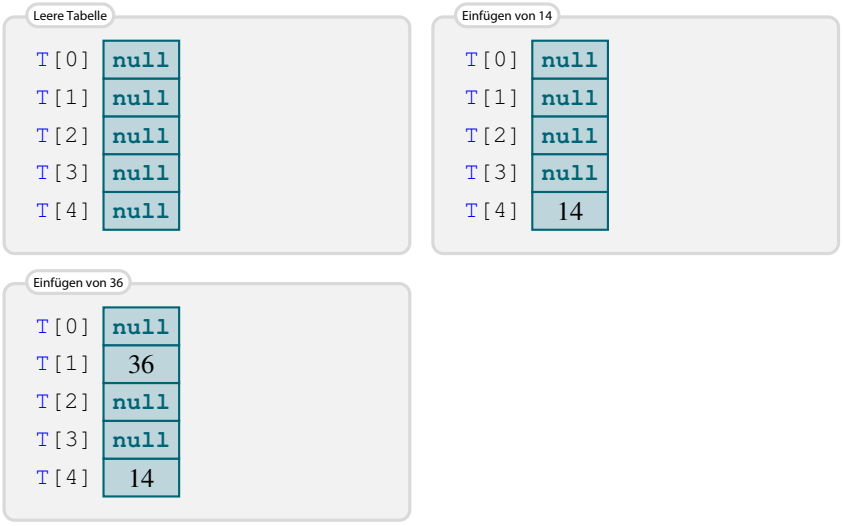
```
interface HashMap <K extends Object, V>
{
  void                insert (K k, V v);
  void                delete (K k);
  HashMap.Entry<K,V> search (K k);
}
```

9.1.1 Die Idee

9-7

Die einfache Grundidee hinter einer Hash-Tabelle.

Eine *Hash-Tabelle* T ist ein Array einer bestimmten Größe r, wobei r in der Regel viel kleiner als N ist. Eine *Hash-Funktion* h bildet die Schlüssel »zufällig« auf Arraypositionen ab, also $h\colon U \to \{0,\ldots,r-1\}$. Ein zu speichernder Schlüssel $x \in U$ wird *idealerweise* an Position $h(x)$ im Array gespeichert.
Beispiel: Einfügen von 14 und 36 mit $h(x) = x \bmod r$.



Will man 9 an Stelle $h(9) = h(14) = 4$ speichern, so *muss man etwas Schlaues* tun. Die Hash-Verfahren unterscheiden sich nur dadurch, was das konkret bedeutet.

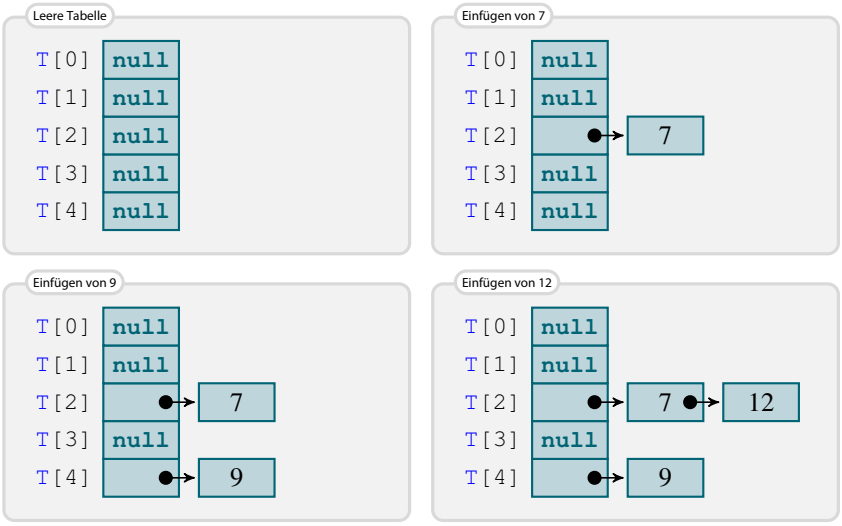
9.1.2 Verkettung

Etwas Schlaues tun I: Verkettung

9-8

- In den Arrayfeldern speichern wir *nicht direkt die Schlüssel*, sondern jedes Arrayfeld ist der Anfang einer *verketteten Liste*.
- Alle Objekte, die von der Hashfunktion auf dasselbe Feld abgebildet werden, werden in der Liste des Feldes gespeichert.
- *Suchen in dieser Liste* ist zwar *langsam*, aber da Kollisionen selten sind, ist die Liste sehr *kurz*.

Einfügen von 7, 9, 12.



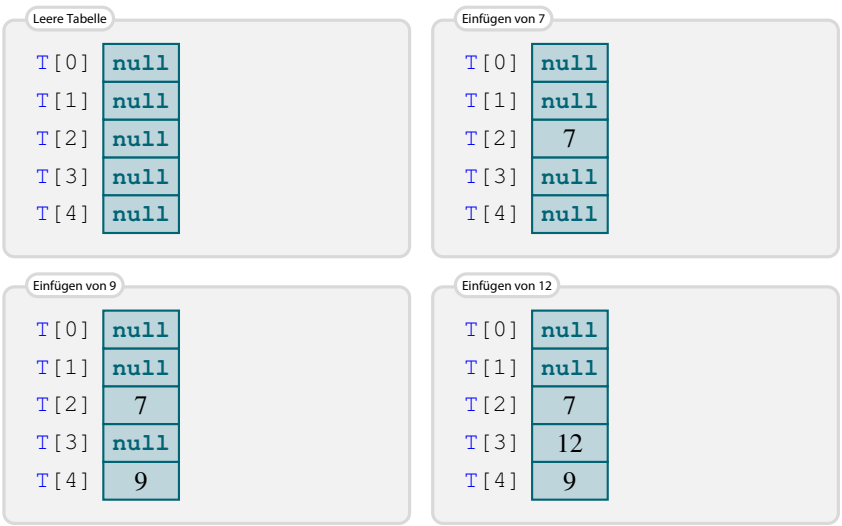
9.1.3 Lineares Sondieren

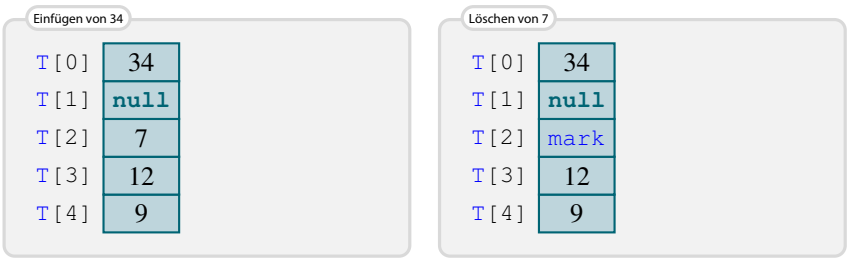
Etwas Schlaues tun II: Lineares Sondieren

9-9

- Ist an $h(x)$ kein Platz, so versuchen wir es eben an Stelle $h(x)+1$.
- Wenn dort kein Platz ist, dann an Stelle $h(x)+2$ und so weiter.
- Beim Suchen müssen wir dann aber neben $h(x)$ auch alle Folgepositionen durchgehen, bis wir eine leere Stelle finden.
- Beim Löschen muss man eine *Lösch-Markierung* an die Stelle setzen.

Einfügen von 7, 9, 12, 34, Löschen von 7.





9.1.4 Hash-Funktionen

9-10 **Hash-Funktionen muss man mit Liebe aussuchen.**

Man muss *Hash-Funktion h sorgfältig wählen*: Da das Universum U viel größer ist als die Tabelle, *muss es notgedrungen sehr viele Kollisionen geben*. Liegen diese Kollisionen *»ungünstig«*, so kommt es zu Verklumpungen.

Beispiel

Sei $r = 256$ die Tabellengröße und $N = 2^{32}$ die Universumsgröße. Sei $h(x) = x \bmod 256$, also das »letzte Byte von x«. Dann ist *die Anzahl der Kollisionen minimal*, aber die Hash-Funktion trotzdem sehr ungeeignet, wenn das letzte Byte aller zu hashenden Schlüssel 0 ist.

9-11 **Universelle Hash-Funktionen haben keine Vorlieben.**

▶ **Definition:** Universelle Hash-Funktionen

Eine Familie H von Hash-Funktionen heißt *universell*, wenn es für je zwei Schlüssel x und y höchstens $|H|/r$ Hash-Funktionen $h \in H$ gibt mit $h(x) = h(y)$.

Wählt man eine Hash-Funktion aus einer universellen Familie H *zufällig*, so ist für je zwei Schlüssel x und y *die Wahrscheinlichkeit einer Kollision nur* $1/r$. Dies ist dieselbe Wahrscheinlichkeit, *als wären* $h(x)$ *und* $h(y)$ *komplett zufällig gewählt*.

▶ **Satz**

Sei $p > N$ eine Primzahl. Dann ist die Familie $H = \{h_{a,b} \mid a,b \in \{1,\ldots,p-1\}\}$ eine universelle Hash-Familie mit

$$h_{a,b}(x) = ((ax+b) \bmod p) \bmod r.$$

Zum (nicht sonderlich schwierigen) Beweis siehe [1].

9.2 Dynamische Größenanpassung

9.2.1 Die Verdoppelung-Halbierungs-Strategie

9-12 **Wie groß sollte eine Hash-Tabelle sein?**

✎ **Zur Diskussion**

Sie rufen `new SuperDuperHashTable()` auf. Dann fügen Sie 10 Elemente ein. Dann suchen Sie 1.000 Mal nach Elementen. Dann fügen Sie 10.000 weitere Elemente ein. Dann suchen Sie 1.000 Mal nach Elementen. Dann löschen Sie 9.999 Element wieder. Wie groß sollte die Hash-Tabelle sein?

9-13 **Der Load-Factor einer Tabelle sollte nicht zu groß und nicht zu klein sein.**

▶ **Definition:** Load-Factor

Der *Load-Factor* einer Hash-Tabelle ist das Verhältnis

$$\alpha = \frac{\text{Anzahl der gespeicherten Elemente}}{\text{Größe der Tabelle (also } r\text{)}}.$$

- Ist α sehr klein (zum Beispiel $\alpha < 0.01$), so wird sehr viel Platz verschwendet.
- Ist α sehr groß ($\alpha > 0.9$), so werden alle Operationen langsam oder unmöglich.

Die Idee des Rehashing.

9-14

Big Idea

Immer, wenn der Load-Factor ein *erlaubtes Intervall* verlässt, so wird die Tabellengröße angepasst, so dass er wieder »gut« ist.

Beispielsweise könnte man verlangen, dass α immer zwischen $\alpha_{\min} = 0.1$ und $\alpha_{\max} = 0.4$ liegt. Wird dies verletzt, so wird die Größe so angepasst, dass wieder $\alpha = \alpha_{\text{ideal}} = 0.25$ gilt.

Die Idee ist gut, aber. . .

Die Größe der Hash-Tabelle hat massiven Einfluss auf die Positionen der Schlüssel. Folglich müssen alle Schlüssel *neu eingefügt werden*.

```
1  algorithm checkLoadFactor
2    r ← Größe von T
3    n ← Anzahl Elemente in T
4    α ← n/r
5    if α < α_min or α > α_max then
6      r_new ← n/α_ideal
7      T_new ← neue Tabelle mit r_new Einträgen
8      // Rehash:
9      for i ← 0 to r − 1 do
10       if T[i] ≠ null and T[i] ≠ mark then
11         insert T[i] into T_new
12     T ← T_new
```

9.2.2 Amortisierte Analyse

Was kostet uns die Dynamik?

9-15

▶ **Satz**

Für eine Hash-Tabelle ohne Größenanpassung mögen die Basis-Operationen je $O(1)$ lange dauern.
Dann kann für die Hash-Tabelle mit Größenanpassung eine Folge von m Einfüge- und Löschen-Operationen höchstens $O(m^2)$ dauern.

Beweis. Offenbar dauert ein Rehash Zeit $O(r)$, maximal also $O(m)$. Da ein Rehash nach jeder der Operationen passieren kann, können m Operationen maximal $O(m^2)$ Aufwand bedeuten. □

✎ **Zur Diskussion**

Vergleichen Sie diese Laufzeit mit der Laufzeit von m Einfüge- und Löschen-Operationen in einen 2-3-Baum.

Die amortisierte Laufzeit der Dynamik.

9-16

Die Worst-Case-Abschätzung scheint *viel zu pessimistisch:* Nach einem Rehash *müssen erst viele Einfüge- oder Löschen-Operationen kommen*, bevor wieder ein Rehash nötig ist. Es liegt nahe, eine *amortisierte Analyse zu versuchen*.

▶ **Satz**

Für eine Hash-Tabelle ohne Größenanpassung mögen die Basis-Operationen je Kosten 1 verursachen.
Dann kann für die Hash-Tabelle mit Größenanpassung mit $\alpha_{\min} < \alpha_{ideal} < \alpha_{\max}$ eine Folge von m Einfüge- und Löschen-Operationen höchstens Kosten $O(m)$ verursachen.

Beweis. Wir zeigen die Behauptung nur für $\alpha_{\min} = 1/8$, $\alpha_{\text{ideal}} = 1/4$ und $\alpha_{\max} = 1/2$; für den allgemeinen Fall siehe Übung 9.1.
Wir definieren eine Potentialfunktion wie folgt: $\Phi(T_i)$ *sei gerade der zwölffache Betrag der Abweichung von der idealen Anzahl der Elemente*, also $\Phi(T_i) = 12|n_i - r_i/4|$. Offenbar ist dies anfangs 0 und nie negativ.
Die amortisierten Kosten des *Einfügens eines Elements* sind:

1. Wenn kein Rehash nötig ist und $n_i/r_i > \alpha_{\text{ideal}}$, so steigt das Potential um 12, also:

$$a_i = \underbrace{1}_{\text{reale Kosten}} + \Phi(T_i) - \Phi(T_{i-1}) = 13.$$

2. Wenn kein Rehash nötig ist und $n_i/r_i < \alpha_{\text{ideal}}$, so fällt das Potential um 12, also: $a_i = 1 + \Phi(T_i) - \Phi(T_{i-1}) = -11$.
3. Wenn ein Rehash nötig ist:

$$a_i = \underbrace{r_i + r_{i-1}}_{\text{reale Kosten}} + \Phi(T_i) - \Phi(T_{i-1}) = \frac{3}{2} r_i + 0 - 12(r_{i-1}/2 - r_{i-1}/4) = 0.$$

Die amortisierten Kosten sind also immer durch eine Konstante beschränkt.
Die amortisierten Kosten des Löschens sind:

1. Wenn kein Rehash nötig ist, wie beim Einfügen gerade -11 oder 13.
2. Wenn ein Rehash nötig ist:

$$\begin{aligned} a_i &= \underbrace{r_i + r_{i-1}}_{\text{reale Kosten}} + \Phi(T_i) - \Phi(T_{i-1}) \\ &= 3r_i + 0 - 12|r_{i-1}/8 - r_{i-1}/4| = 0. \end{aligned}$$

Wieder sind die amortisierten Kosten also konstant.
Die amortisierten Kosten von m Operationen sind folglich höchstens $13m$ und somit auch die realen Kosten. □

Zusammenfassung dieses Kapitels

▶ Hash-Tabellen mit veränderlicher Größe

Passt man die Größe von Hash-Tabellen immer dann an, wenn der Load-Factor Schwellwerte *über-* oder *unterschreitet*, so sind bleiben die *amortisierte Kosten konstant*.

Zum Weiterlesen

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms,* zweite Auflage, MIT Press, 2001, Kapitel »Hash Tables«.

Übungen zu diesem Kapitel

Übung 9.1 Amortisierte Analyse von Hash-Tabellen mit Größenanpassung, mittel

Beweisen Sie Satz 9-16 für beliebige $\alpha_{\min}$, α_{ideal} und $\alpha_{\max}$.

Tipp: Passen Sie die Potentialfunktion aus dem Beweis geeignet an.

Übung 9.2 Hashing mit dynamischer Größenanpassung, mittel

Im Folgenden soll eine Hashtabelle mit linearer Sondierung, dynamischer Größenanpassung und einer universellen Familie $H_p = \{h_{a,b} \mid a,b \in \{1,\ldots,p-1\}\}$ mit $h_{a,b}(x) = ((ax+b) \bmod p) \bmod r$ von Hashfunktionen simuliert werden. Das dynamische Verhalten der Tabelle ist durch die Parameter $\alpha_{\min} = 1/8$, $\alpha_{\text{ideal}} = 1/4$, und $\alpha_{\max} = 1/2$ beschrieben.

1. Starten Sie mit einer Tabelle der Größe $r = 4$ und fügen Sie nacheinander Elemente mit den Schlüsseln 3, 5, 1 und 2 ein. Verwenden Sie hierzu die Hash-Funktion $h_{10,16} \in H_{31}$. Hierbei wird eine Größenanpassung notwendig, verwenden Sie nach der Größenanpassung die Funktion $h_{7,8} \in H_{31}$.
2. Löschen Sie aus der Tabelle nun die Werte 2 und 5. Geben Sie dabei explizit die Berechnung der Hashwerte und den gesamten Inhalt der Tabelle nach jeder Löschoperation an.

Übung 9.3 Kuckuck-Hashing simulieren, mittel

In dieser Aufgabe werden wir das Kuckuck-Hashing an einem Beispiel simulieren. Hierbei verwenden wir die Hashfunktion $h_{11,14} \in H_{17}$ für die erste der beiden Tabellen und die Hashfunktion $h_{1,10} \in H_{17}$ für die zweite. Eine dynamische Größenanpassung soll in dieser Aufgabe erst einmal nicht vorgenommen werden.

Starten Sie mit zwei leeren Tabellen der Größe $r = 3$ für das Kuckuck-Hashing. Fügen Sie nacheinander die Elemente 3, 5, 2, 7 und 10 ein.

10-1

Kapitel 10
A&D: Perfektes Hashing

Kuckucks-Hash-Tabellen sind perfekt, dynamisch und verdammt schnell

10-2

Lernziele dieses Kapitels

1. Idee des perfekten Hashings und einfache perfekte Hash-Verfahren kennen
2. Kuckucks-Hash-Tabellen verstehen und implementieren können

Inhalte dieses Kapitels

Worum es heute geht

Die bekanntesten Hash-Tabellen sind sichlicher die Hash-Tabellen mit linearer Sondierung: Ist eine Stelle in der Tabelle schon voll, so sucht man sich eben den nächsten leeren Platz. Diese Methode ist in der Praxis ziemlich gut (genaugenommen ist sie fast unschlagbar gut), jedoch gibt es aus theoretischer und (ganz selten) auch aus praktischer Sicht einen gravierenden Nachteil: Es *kann* sein, dass es recht lange dauert, ein Element zu finden. Das ist zwar sehr unwahrscheinlich, aber eben auch nicht unmöglich.

Aus diesem Grund hat man lange geforscht, eine Variante von Hash-Tabellen zu finden, die genauso schnell ist wie Hash-Tabellen mit linearer Sondierung, die aber eine *garantierte* konstante Suchzeit hat. Mit dieser Problematik haben sich dann Theoretiker extensiv beschäftigt und auch Erfolg gehabt. Jedoch hatte dieser »Erfolg« einen Schönheitsfehler: die entwickelten »Dynamic Amortized Perfect Hash Tables« haben alle nur denkbaren schönen theoretischen Eigenschaften – in der Praxis sind Hash-Tabellen mit linearer Sondierung trotzdem viel schneller. Das liegt daran, dass diese Tabellen intern doch reichlich komplex aufgebaut sind und der Verwaltungsoverhead enorm ist.

Das Blatt hat sich 2001 gewandelt, als auf dem European Symposium on Algorithms das Kuckucks-Hashing vorgestellt wurde. Diese Methode hat drei entscheidende Vorteile:

1. Sie ist in der Praxis fast genauso schnell wie lineares Sondieren.
2. Sie garantiert, dass die Suche nie mehr als zwei Speicherzugriffe braucht (und damit immer eine sehr kleine konstante Zeit braucht).
3. Sie ist sehr einfach zu verstehen und zu implementieren.

Damit ist Kuckucks-Hashing eine praktische Methode des Hashings, die aber auch die Theoriegemeinde glücklich gemacht hat: Wie wir in diesem Kapitel sehen werden, ist die Analyse des Kuckucks-Hashing durchaus interessant und deutlich komplexer als die eigentliche Datenstruktur (das haben sie dann mit der Union-Find-Datenstruktur gemeinsam).

10.1 Perfektes Hashing: Die Anforderung

Was sollte die perfekte Hash-Tabelle leisten?

10-4

Das *Suchen nach einem Schlüssel* ist in einer Hash-Tabelle sicherlich die *häufigste* Operation. Man kann zeigen, dass diese *im Schnitt* dauern:

- $O(1+\alpha)$ bei Hash-Tabellen mit Verkettung und
- $O(\frac{1}{1-\alpha})$ bei Hash-Tabellen mit linearem Sondieren.

In beiden Fällen ist jedoch die *Worst-Case-Zeit* $O(n)$, wenn es n Elemente gibt.

▶ Definition: Perfekte Hash-Verfahren

Ein Hash-Verfahren heißt *perfekt*, wenn die Zeit für eine Suche *im Worst-Case* $O(1)$ beträgt.

(Einfüge- und Löschen-Operationen können hingegen länger dauern.)

10.2 Statische perfekte Hash-Tabellen

10.2.1 Geburtstagstabellen

Der statische Fall.

10-5

Wir wollen zunächst einige *recht starke Voraussetzungen* machen:

1. Die Menge der n zu hashenden Werte ist fest (statisch) und von vornherein bekannt.
2. Speicherplatz ist kein Problem.

Beispiele

- Schlüsselworte in einer Programmiersprache.
- Erlaubte XML-Tags aufgrund einer Document-Type-Description.
- Optionen eines Shell-Befehls.

Interludium: Das Geburtstagsparadoxon.

10-6

✎ Zur Diskussion

Wie hoch ist die Wahrscheinlichkeit, dass zwei der Anwesenden im Hörsaal am gleichen Tag des Jahres Geburtstag haben?

- Bei 23 Anwesenden ist sie höher als 50%.
- Bei 50 Anwesenden ist sie höher als 97%.
- Bei 57 Anwesenden ist sie höher als 99%.
- Bei 100 Anwesenden ist sie höher als 99,99997%.

Allerdings:

- Bei 10 Anwesenden ist sie unter 12%.
- Bei 20 Anwesenden ist sie unter 42%.

Interludium: Das Geburtstagsparadoxon auf dem Mars

10-7

✎ Zur Diskussion

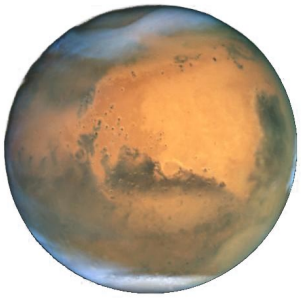
Wie hoch ist die Wahrscheinlichkeit, dass zwei der Anwesenden im Hörsaal am gleichen Tag des *Mars-Jahres* Geburtstag haben, *wenn die Vorlesung auf dem Mars stattfinden würde*?

Das Mars-Jahr hat 687 Tage.

- Bei 32 Anwesenden ist sie höher als 50%.

Allerdings:

- Bei 10 Anwesenden ist sie unter 10%.
- Bei 20 Anwesenden ist sie unter 26%.



NASA, Public domain

10-8

Wie man das Geburtstagsparadoxon in ein Hash-Verfahren umwandelt.

Idee zum perfekten statischen Hashing

Wären die *Geburtstage* gerade die Hashwerte und wären die *Tage des Jahres* die möglichen Tabelleneinträge, so erhält man *mit über 50% Wahrscheinlichkeit ein Hashing ohne Kollisionen*, wenn es nur 22 zu hashende Studenten gibt.

Allgemein gilt:

▶ **Satz**

Ist $r = n^2$, so gibt es bei einer zufälligen Wahl der Hash-Funktion mit Wahrscheinlichkeit mindestens 50% keine Kollisionen.

Beweis. Die Wahrscheinlichkeit für eine Kollision von zwei Werten ist $1/r = 1/n^2$. Es gibt $\binom{n}{2}$ Paare. Der Erwartungswert für die Anzahl X an Kollisionen ist somit $\mathrm{E}[X] = \binom{n}{2}\frac{1}{n^2} = \frac{1}{2}\frac{n^2-n}{n^2} < 1/2$. Nach der Markov-Ungleichung gilt $\Pr[X \geq t] \leq \mathrm{E}[X]/t$ und für $t = 1$ ist also die Wahrscheinlichkeit, dass es mindestens eine Kollision gibt, kleiner als $1/2$. □

Merke

Sind in einem Hörsaal n Studierende auf einem Planeten mit Umlaufzeit n^2 Tagen, so haben mit weniger als 50% Wahrscheinlichkeit zwei am selben Tag des Planeten-Jahres Geburtstag.

10-9

Geburtstagstabellen: perfektes statisches Hashing.

```
1  algorithm build_birthday_table(k_1, v_1, ..., k_n, v_n)
2    p ← pick a prime larger than N
3    start:
4    T ← new Entry [n^2]
5    a ← random value between 1 and p − 1
6    b ← random value between 1 and p − 1
7    for i ← 1 to n do
8      if T[h_{a,b}(k_i)] is empty then
9        T[h_{a,b}(k_i)] ← (k_i, v_i)
10     else
11       goto start
12   return (T, p, a, b)
13
14 algorithm search(key)
15   e ← T[h_{a,b}(key)]
16   if e is not null and e.key = key then
17     return e
18   else
19     return null
```

10-10

Und jetzt noch Platz sparen. . .

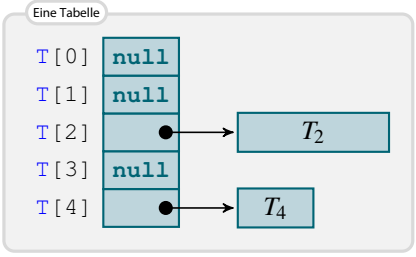
Das perfekte Hashing mit einer Tabelle der Größe n^2 funktioniert schon ganz gut, aber der Platzbedarf ist für größere n viel zu groß.

Der Trick mit den zweistufigen Tabellen

Um Platz zu sparen, gehen wir wie folgt vor: Wir benutzen zunächst doch eine Tabelle der Größe $r = n$, wodurch (fast) mit Sicherheit viele Kollisionen zustande kommen. Deshalb speichern wir für jede Stelle alle Elemente, die auf diese Stelle gehasht werden, *in einer (kleinen) perfekten Geburtstagstabelle*. Diese Tabellen haben zwar *quadratische Größe in der Anzahl ihrer Einträge*, jedoch *gibt es ja auch nur wenige Einträge*. Die Analyse zeigt gleich, dass *mit hoher Wahrscheinlichkeit die Summe der Größen der Geburtstagstabelle* lediglich $O(n)$ ist.

Der Aufbau der statischen perfekten Hash-Tabelle.

10-11



Die Haupttabelle links mit Hashfunktion h speichert Verweise auf viele kleine Geburtstagstabellen T_i. Jede speichert:

1. Alle Schlüssel k mit $h(k) = i$. Seien dies n_i viele.
2. Die (quadratische) Größe $r_i = n_i^2$.
3. Die zufällig gewählten Zahlen a_i und b_i für eine lokale Hashfunktion h_{a_i,b_i} für diese Tabelle.

Offenbar benötigt die Gesamttabelle insgesamt Platz

$$\sum_{i=0}^{r-1} n_i^2.$$

Die Algorithmen hinter der statischen perfekten Hash-Tabelle.

10-12

```
1  algorithm build_perfect_table(k_1, v_1, ..., k_n, v_n)
2    T ← new Table [n]
3    h ← random hash function for T (pick h = h_{a,b} for random a and b)
4
5    L ← new List [n]
6    for i ← 1 to n do
7      append (k_i, v_i) to L[h(k_i)]
8
9    for i ← 1 to n do
10     T[i] ← build_birthday_table(L[i])
11
12 algorithm search(key)
13   return T[h(key)].search(key)
```

10.2.2 Analyse der statischen perfekten Hash-Tabellen

Viele Geburtstagstabellen brauchen weniger Platz als nur eine.

10-13

▶ Satz

Der Erwartungswert für die Gesamtgröße der statischen perfekten Hash-Tabelle ist

$$\mathrm{E}\left[\sum_{i=0}^{r-1} n_i^2\right] < 2n.$$

Beweis. Da $n_i^2 = n_i + 2\binom{n_i}{2}$ allgemein gilt, lässt sich der Erwartungswert wie folgt umschreiben:

$$\mathrm{E}\left[\sum_{i=0}^{r-1} n_i^2\right] = \mathrm{E}\left[\underbrace{\sum_{i=0}^{r-1} n_i}_{=n}\right] + 2\,\mathrm{E}\left[\sum_{i=0}^{r-1} \binom{n_i}{2}\right].$$

Die Summe $\sum_{i=0}^{r-1}\binom{n_i}{2}$ ist die *Anzahl aller Kollisionen*. Da wir universelles Hashing benutzen, ist die *Wahrscheinlichkeit einer Kollision gerade* $1/r = 1/n$. Folglich ist der Erwartungswert für die Anzahl an Kollisionen gerade $\binom{n}{2}\frac{1}{n} = \frac{n-1}{2}$. Setzt man dies oben ein, so erhält man die Behauptung. □

10-14

Zusammenfassung zum statischen perfekten Hashing.

Um n Werte in einer *statischen* perfekten Hash-Tabelle zu speichern, geht man wie folgt vor:

1. Wähle eine Hash-Funktion h zufällig.
2. Ermittle für jedes $i \in \{0,\ldots,r-1\}$, wie viele Elemente n_i hierauf gehasht würden.
3. Ist die Summe $S = \sum_{i=0}^{r-1} n_i^2$ größer als $4n$, so wähle eine andere Hash-Funktion und starte neu. (Die Wahrscheinlichkeit hierfür ist nach der Markov-Ungleichung nur $\Pr[S \geq 4n] \leq \mathrm{E}[S]/4n < 1/2$.)
4. Anderenfalls baue mit dem Algorithmus *build_perfect_table* die perfekte Hash-Tabelle.
5. Die resultierende Tabelle hat *Gesamtgröße* $4n$ und *Suchen benötigt immer und somit auch im Worst-Case zwei Speicher-Zugriffe*.

10.3 Kuckucks-Hashing

10.3.1 Die Idee

10-15

Ideen muss man haben.

Die Situation bis 2001

Die Idee der *statischen* perfekte Hash-Tabelle kann man zu einer *dynamischen* perfekten Hash-Tabelle weiterentwickeln. Dabei kommt dann allerdings eine *komplexe Datenstruktur heraus*, die *in der Praxis untauglich* ist.

Das Kuckucks-Hashing

Im Jahr 2001 wurde dann aber ein *völlig neues Verfahren* für das dynamische perfekte Hashing präsentiert, das *völlig anders ist* als alle bisherigen und *extrem simple* ist.

10-16

Eine geniale Idee.

Das Kuckucks-Hashing baut auf folgenden Grundideen auf:

1. Es gibt *zwei Hash-Tabellen* mit *zwei Hash-Funktionen* statt nur einer.
2. Jedes Element ist *entweder in der einen oder in der anderen Tabelle* an seinem Hash-Wert. (Hierdurch wird die Tabelle perfekt.)
3. Fügt man ein Element ein und sind in beiden Tabellen die Einträge schon belegt, so *fügt man das Element trotzdem ein* und dafür *fliegt ein Element raus* (Kuckucks-Operation), das dann aber *in die andere Tabelle kommt*. Ist doch auch kein Platz, so wiederholt sich das Spiel.

10.3.2 Die Implementation

10-17

Der Aufbau im Detail.

▶ **Definition**

Eine Kuckucks-Hash-Tabelle besteht aus zwei normalen Hash-Tabellen T_1 und T_2 mit folgenden Eigenschaften:

1. Jede Tabelle hat mindestens $r = 2n$ Einträge (es würde aber schon $r = (1+\varepsilon)n$ reichen).
2. Die zugehörigen Hash-Funktionen h_1 und h_2 sind zufällig und unabhängig.
3. Jeder Schlüssel k ist entweder an der Stelle $T_1[h_1(k)]$ gespeichert oder an $T_2[h_2(k)]$.

Beispiel: Kuckucks-Hash-Tabelle

Seien $h_1(x) = (2x \bmod 7) \bmod 5$ und $h_2(x) = (3x \bmod 7) \bmod 5$.

T1[0]		T2[0]	
T1[1]	4	T2[1]	
T1[2]	1	T2[2]	3
T1[3]		T2[3]	
T1[4]	2	T2[4]	

Suchen und Löschen sind wirklich einfach.

10-18

```
1  algorithm search(key)
2    e ← T1[h1(key)]
3    if e is not null and e.key = key then
4      return e
5    else
6      e ← T2[h2(key)]
7      if e is not null and e.key = key then
8        return e
9      else
10       return null
11
12 algorithm delete(key)
13   e ← T1[h1(key)]
14   if e is not null and e.key=key then
15     T1[h1(key)] ← null
16   else
17     e ← T2[h2(key)]
18     if e is not null and e.key=key then
19       T2[h2(key)] ← null
```

Offenbar brauchen beide Operationen nur konstante Zeit (und sind auch praktisch sehr schnell).

Das Einfügen und der Kuckuck.

10-19

Beim *Einfügen* kann es passieren, dass ein Element *weder in die erste noch in die zweite Tabelle an den gewünschten Platz passt*. In diesem Fall *schafft die Kuckucks-Operation Platz:*

Kuckucks-Operation

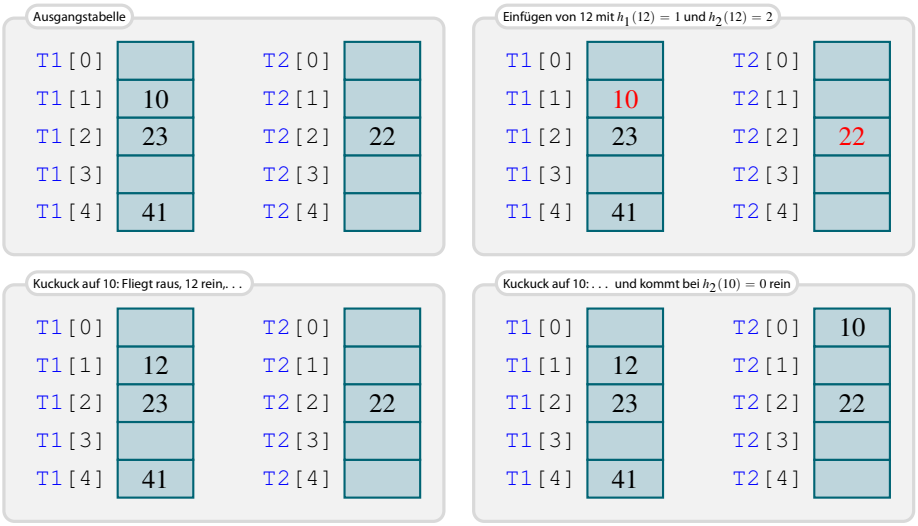
Sei $i \in \{1,2\}$ und $j = 3 - i$ »der Index der anderen Tabelle«. Nehmen wir an, in T_i ist an Stelle $h_i(k)$ ein Element mit Schlüssel k gespeichert. Die *Kuckucks-Operation* entfernt nun dieses Element aus T_i und fügt es an der Stelle $h_j(k)$ in der Tabelle T_j ein.

Offenbar kann es bei der Kuckucks-Operation vorkommen, dass am »Zielort $T_j[h_j(k)]$» wieder kein Platz ist. Dann führen wir wieder die Kuckucks-Operation hierauf aus, bis wir einen Platz finden.
Sollte man in eine Endlosschleife geraten, so muss man sich etwas Schlaues ausdenken.

Beispiel eines Einfügevorgangs.

10-20

Seien $h_1(x) = (x \bmod 100) \operatorname{div} 10$ und $h_2(x) = x \bmod 10$.



Direktes Einfügen von 123

	T1	T2
[0]		10
[1]	12	
[2]	23	22
[3]		123
[4]	41	

Einfügen von 43

	T1	T2
[0]		10
[1]	12	
[2]	23	22
[3]		123
[4]	41	

Kuckuck auf 41: Fliegt raus, 43 rein,. . .

	T1	T2
[0]		10
[1]	12	
[2]	23	22
[3]		123
[4]	43	

Kuckuck auf 41:. . . und kommt bei $h_2(41) = 1$ rein

	T1	T2
[0]		10
[1]	12	41
[2]	23	22
[3]		123
[4]	43	

Einfügen von 110

	T1	T2
[0]		10
[1]	12	41
[2]	23	22
[3]		123
[4]	43	

Kuckuck auf 13: Fliegt raus, 110 rein,. . .

	T1	T2
[0]		10
[1]	110	41
[2]	23	22
[3]		123
[4]	43	

Kuckuck auf 13:. . . und kommt bei $h_2(12) = 2$ rein

	T1	T2
[0]		10
[1]	110	41
[2]	23	22
[3]		123
[4]	43	

Kuckuck auf 22: Fliegt raus, 12 rein,. . .

	T1	T2
[0]		10
[1]	110	41
[2]	23	12
[3]		123
[4]	43	

Kuckuck auf 22:. . . und kommt bei $h_1(22) = 2$ rein

	T1	T2
[0]		10
[1]	110	41
[2]	23	12
[3]		123
[4]	43	

Kuckuck auf 23: Fliegt raus, 22 rein,. . .

	T1	T2
[0]		10
[1]	110	41
[2]	22	12
[3]		123
[4]	43	

Kuckuck auf 23:. . . und kommt bei $h_2(23) = 3$ rein

	T1	T2
[0]		10
[1]	110	41
[2]	22	12
[3]		123
[4]	43	

Kuckuck auf 123: Fliegt raus, 23 rein,. . .

	T1	T2
[0]		10
[1]	110	41
[2]	22	12
[3]		23
[4]	43	

Kuckuck auf 123:. . . und kommt bei $h_1(123) = 2$ rein

	T1	T2
[0]		10
[1]	110	41
[2]	22	12
[3]		23
[4]	43	

Pseudo-Code des Einfügevorgangs.

10-21

```
1  algorithm insert(k, v)
2    insert_me ← (k, v)
3    if T1[h1(k)] is empty then
4      T1[h1(k)] ← insert_me
5    else if T2[h2(k)] is empty then
6      T2[h2(k)] ← insert_me
7    else
8      count_down ← n
9      i ← 1
10     swap (insert_me, Ti[hi(insert_me.k)])
11     while insert_me is not null and count_down >0 do
12       i ← 3 − i
13       swap (insert_me, Ti[hi(insert_me.k)])
14       count_down ← count_down -1
15     if insert_me is not null then
16       rehash
17       insert(insert_me)
```

Zusammenfassung zur Implementation von Kuckucks-Hash-Tabelle

10-22

Laufzeit der Grundoperationen

1. Einfüge-Operationen benötigen immer maximal zwei Zugriffe.
2. Löschen-Operationen benötigen ebenfalls maximal zwei Zugriffe.
3. Einfüge-Operationen können, sogar mehrfach, zu einem Rehash der gesamten Tabelle führen.

Effekt der Größenanpassung

Es kann zu einem *Überlauf* oder *Unterlauf* der Tabelle kommen:

1. Wir setzen $\alpha_{\max} = 1/16$, (wobei allerdings schon $\frac{1}{2(1+\varepsilon)}$ reichen würde). Falls also die Tabellen insgesamt mehr als $r/8$ Elemente speichern sollen, werden sie vergrößert und ein Rehash durchgeführt.
2. Man kann $\alpha_{\min}$ beliebig setzen. Ein Wert von beispielsweise $1/64$ bietet sich an.

Wir haben schon gesehen, dass die *amortisierten Kosten hiervon konstant* sind.

10.3.3 Analyse der dynamischen perfekten Hash-Tabellen

Wie gut funktioniert das Kuckucks-Hashing?

10-23

▶ Satz

Sei eine Folge von Einfüge-, Such- und Löschen-Operationen gegeben. Dann benötigen diese Operationen folgende Zeiten:

1. *Alle Such-Operationen benötigen immer maximal Zeit $O(1)$.*
2. *Die Löschen-Operationen benötigen amortisierte Zeit $O(1)$. Wenn $\alpha_{\min} = 0$, so benötigen sie sogar immer maximal Zeit $O(1)$.*
3. *Die Einfüge-Operationen benötigen erwartete amortisierte Zeit $O(1)$.*

Die ersten beiden Punkte sollten klar sein. Im Folgenden soll also der dritte Punkte bewiesen werden und wir nehmen *vereinfachend an, dass einfach nur n Elemente in zwei leere Tabelle der Größe jeweils $r = 2n$ eingefügt werden.*

10-24

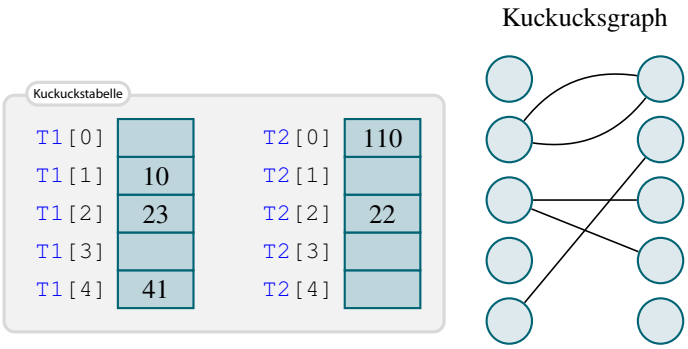
Erste Vorbereitung: Der »Kuckucksgraph«

Für die Analyse ist offenbar wichtig, *wie Elemente entlangt von »Kanten« zwischen den beiden Tabellen wechseln.* Wir führen daher Begriffe ein, die das »modellieren«:

▶ **Definition:** Kuckucksgraph

Seien T_1 und T_2 zwei zum Teil gefüllte Kuckuckstabellen. Der zugehörige *Kuckucksgraph* ist ein ungerichteter Multigraph (es kann Kanten mehrfach geben) mit $2r$ Knoten, nämlich je einen pro »Slot« der beiden Tabellen, und für jeden der n nichtleeren Einträge x in einer der beiden Tabellen einer ungerichteten Kante zwischen dem Knoten, der Slot $h_1(x)$ der ersten Tabelle entspricht, und dem Knoten, der Slot $h_2(x)$ der zweiten Tabelle entspricht.

Beispiel



10-25

Zweite Vorbereitung: Der »Kuckuckspfad«.

▶ **Definition:** Kuckuckspfad

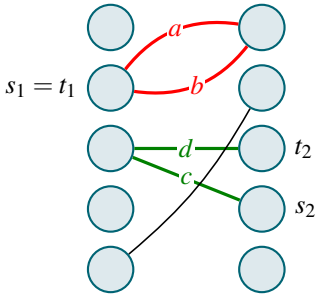
Ein *Kuckuckspfad von s nach t* ist eine *nichtleere* Folge $e_1, \ldots, e_k$ von *unterschiedlichen* Kanten im Kuckucksgraph, so dass

1. die erste Kante s enthält und die letzte t und
2. je zwei aufeinander folgende Kante einen Knoten gemeinsam haben.

Bemerkung: »Unterschiedliche« Kanten bedeutet, dass die Kanten im Multigraphen unterschiedlich sein müssen, wohl aber zwischen denselben Knoten verlaufen können.

Beispiel

Die roten Kanten (a,b) bilden einen Kuckuckspfad von s_1 nach t_1, die grünen Kanten (c,d) einen von s_2 nach t_2. Die Folgen (a,b,c) und (a,b,a) bilden keine Kuckuckspfade:



10-26

Die zwei Schritte der Analyse: Die erwartete Laufzeit ist immer $O(1)$.

Wir zeigen, dass in den beiden folgenden Fällen die *erwartete Laufzeit für das Einfügen eines Elementes x nur $O(1)$ beträgt:*

1. Das Einfügen »gelingt«.
2. Das Einfügen »misslingt« und wir müssen (mindestens) einmal rehashen.

10-27

Analyse des Falls »das Einfügen gelingt«.

Falls das Einfügen von *x gelingt,* so muss Folgendes geschehen sein: Beginnend bei einem Knoten im Kuckucksgraph sind Elemente *immer entlang von Kanten des Graphen verschoben worden,* bis irgendwann Platz für das letzte Element war.

Beobachtung 1

Alle Verschiebungen waren entlang von Kanten *innerhalb einer Zusammenhangskomponente* des Kuckucksgraphen.

Beobachtung 2
Entlang einer Kante kann ein Element höchstens zweimal verschoben werden (sonst gerät man nämlich in eine Endlosschleife).

▶ Folgerung
Die erwartete Größe der Zusammenhangskomponenten ist eine obere Schranke für die erwartete Laufzeit im Fall »das Einfügen gelingt«.

Das kleine Pfadlängenlemma.

10-28

▶ Lemma
Seien s und t Knoten in einem Kuckucksgraphen. Dann ist die Wahrscheinlichkeit, dass es im Kuckucksgraphen eine Kante von s nach t gibt, höchstens $1/(2r)$.

Beweis.
Für eine beliebige Kante x gilt: Die Wahrscheinlichkeit für $h_1(x) = s$ und für $h_2(x) = t$ beträgt jeweils $1/r$. Insgesamt ist die Wahrscheinlichkeit, dass x gerade s und t verbindet, höchstens $1/r^2$. Da es $n = r/2$ Kanten gibt, verbindet mit Wahrscheinlichkeit höchstens $n/r^2 \leq 1/(2r)$ eine von ihnen gerade s und t. □

Das große Pfadlängenlemma.

10-29

▶ Lemma
Seien s und t Knoten in einem Kuckucksgraphen und $l \geq 1$ eine Länge. Dann ist die Wahrscheinlichkeit, dass der kürzeste Kuckuckspfad von s nach t genau Länge l hat, höchstens $1/(2^l r)$.

Beweis durch Induktion über l. Für $l = 1$ liefert das kleine Pfadlängenlemma die Behauptung. Für Schritt von $l-1$ zu l gibt es genau dann einen kürzesten Pfad Kuckuckspfad von s nach t, wenn es einen Knoten x gibt mit:

1. Es gibt einen kürzesten Kuckuckspfad von s nach x der Länge $l-1$.
2. Es gibt eine (weitere) Kante zwischen von x nach t.

Nach Induktionsvoraussetzung ist die Wahrscheinlichkeit für den ersten Punkt $1/(2^{l-1}r)$ für ein festes x und $1/(2r)$ für den zweiten, was zusammen $1/(2^l r^2)$ ergibt.
Da es r mögliche x gibt (alle in der t gegenüberliegenden Seite), folgt die Behauptung. □

Die erwartete Größe der Zusammenhangskomponenten.

10-30

▶ Lemma
Die erwartete Größe einer Zusammenhangskomponente des Kuckucksgraphen ist $O(1)$.

Beweis. Zwei unterschiedliche Knoten x und y liegen höchstens mit Wahrscheinlichkeit $1/r$ in derselben Zusammenhangskomponente: Es muss dafür einen Weg irgendeiner Länge $l \geq 1$ geben, was mit Wahrscheinlichkeit $\sum_{l=1}^{\infty} 1/(2^l r) = 1/r$ der Fall ist. Hieraus folgt sofort, dass die *erwartete Anzahl weiterer Elemente, die in derselben Komponente wie ein gegebenes x sind,* höchstens $2r/r = 2$ sind. □

Damit ist gezeigt, dass der Fall »das Einfügen gelingt« erwartete Zeit $O(1)$ dauert.

Analyse des Falls »das Einfügen misslingt«.

10-31

Zunächst eine einfache, aber entscheidende Beobachtung:

Beobachtung
Der Fall »das Einfügen misslingt« kann nur auftreten, wenn es einen Kreis im Kuckucksgraphen gibt, also einen Kuckuckspfad der Länge $l \geq 2$ von einem Knoten s zu s der ersten Tabelle.

Nun gilt:

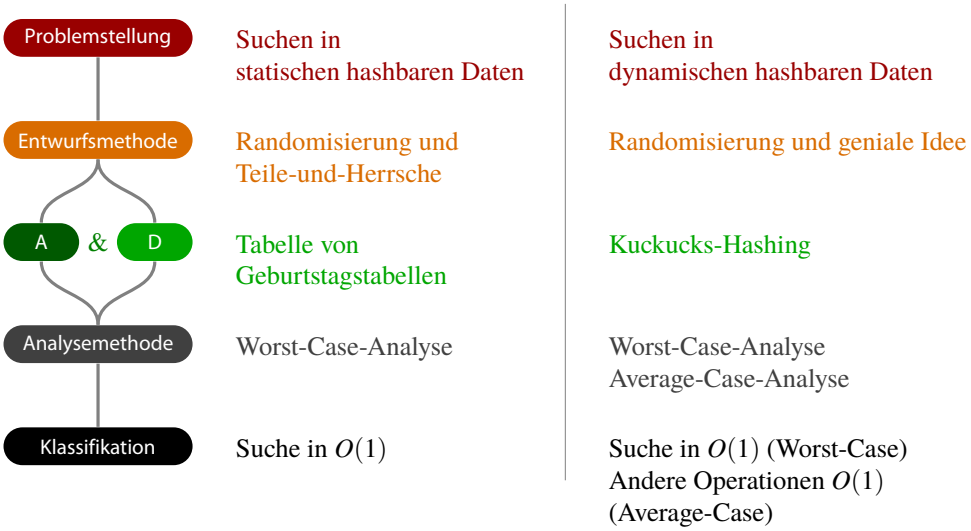
- Für ein konkretes s ist die Wahrscheinlichkeit hiervon aber nach dem Pfadlängenlemma höchstens $1/(2^l r) \leq 1/(4r)$; summiert über alle möglichen s also höchstens $r/(4r) = 1/4$.
- Es gibt also *nur mit Wahrscheinlichkeit* $1/4$ *überhaupt einen Rehash innerhalb der n Einfüge-Operationen.*

- Analog gibt es dann *nur mit Wahrscheinlichkeit* $1/4^2$ *zwei Rehashes innerhalb der* n *Einfüge-Operationen*
- und so weiter für i Rehashes mit Wahrscheinlichkeit $1/4^i$.

Die *erwartete Anzahl Rehashes* während der n Einfüge-Operationen ist also höchstens $\sum_{i=1}^{\infty} i/4^i = O(1)$.
Da jeder Rehash höchstens $O(n)$ lange dauert, sind die erwarteten amortisierten Rehash-Kosten von n Einfüge-Operationen $O(n)$ und damit $O(1)$ pro Operation.

Zusammenfassung dieses Kapitels

10-32



▶ Perfektes Hashing

Ein Hash-Verfahren ist *perfekt*, wenn die Such-Operationen im *Worst-Case* $O(1)$ dauert. Statische perfekte Hash-Verfahren sind:

- Geburtstagstabellen
- Tabellen von Geburtstagstabellen

Letztere benötigen auch nur Platz $O(n)$.

▶ Kuckucks-Hashing

Das *Kuckucks-Hashing* ist ein *dynamisches perfektes* Hash-Verfahren. Die Grundideen sind:

1. Es gibt zwei Hash-Tabellen.
2. Jedes Element ist an einer seiner Hash-Stellen in einer der Tabellen.
3. Passt ein neues Element an keinen der Plätze, wird eines der dort befindlichen Elemente verdrängt (Kuckucks-Operation), das dann wieder ein Element in der anderen Tabelle verdrängt und so fort.

Zum Weiterlesen

[1] Rasmus Pagh and Flemming F. Rodler, Cuckoo Hashing, *Journal of Algorithms*, 51(2):122–144, 2004.

In dieser Arbeit wird das Kuckucks-Hashing eingeführt und analysiert. Diese Arbeit ist auch für Nicht-Theoretiker sehr gut lesbar, da das Verfahren eben nicht nur theoretisch erörtert wird, sondern auch ausführlichen praktischen Vergleichstests unterzogen wurde. Laut den Autoren scheint dies überhaupt die erste Arbeit zu sein, in der eine Reihe von Standard-Hashverfahren auf einer aktuellen Rechnerarchitektur verglichen wurden. Dabei fördern die Autoren ebenso spannende wie überraschende Ergebnisse zu Tage: Beispielsweise schneidet das recht triviale lineare Sondieren durch die Bank am besten ab, was an der besonders guten Ausnutzung der Cache-Architektur moderner Prozessoren liegt. Im Vergleich dazu ist beispielsweise Hashing mit Verkettung viel schlechter. Kuckucks-Hashing hält sich recht wacker und kommt bis auf 20% bis 30% an die Geschwindigkeit des linearen Sondierens heran, garantiert dafür aber auch perfektes Hashing (also nur zwei Zugriffe beim Suchen im Worst-Case). Dynamische Varianten von Verfahren, die auf Geburtstagstabellen aufbauen, sind mit Faktor 6 aufwärts so weit abgeschlagen, dass sie gar nicht untersucht wurden.

Übungen zu diesem Kapitel

Übung 10.1 Hashing-Verfahren implementieren, mittel bis schwer

In dieser Aufgabe soll das Hashing mit linearer Sondierung und Kuckucks-Hashing jeweils mit dynamischer Größenanpassung in Java implementiert werden. Gehen Sie dabei wie folgt vor:

1. Erstellen Sie zwei Klassen, die jeweils das Interface

```
public interface Hashfunction {
  int hash(int key);
}
```

   implementieren. Eine Klasse `SimpleHash` soll zur Berechnung der einfachen Hashfunktion $h(x) = x \bmod r$ verwendet werden können, die andere Klasse `UniversalHash` soll Hashfunktionen der Familie H_p repräsentieren.
2. Implementieren Sie Hashing mit linearer Sondierung zunächst ohne dynamische Größenanpassung. Kapseln Sie die Implementierung in eine Klasse, die das Interface

```
public interface HashMap {
  void insert(int key);
  void delete(int key);
  boolean search (int key);
}
```

   implementiert. Objekte dieser Klassen repräsentieren Hashtabellen, die auf linearer Sondierung basieren. Erweitern Sie Ihre Klasse um eine Methode `rehash`, die beim Einfügen und Löschen aufgerufen wird, um die Größe der Tabelle anzupassen. Sie können annehmen, dass $\alpha_{\text{min}} = 1/8$, $\alpha_{\text{ideal}} = 1/4$ und $\alpha_{\text{max}} = 1/2$ gilt.
3. Implementieren Sie Kuckuck-Hashing zunächst ohne dynamische Größenanpassung in einer Klasse, die das Interface `HashMap` implementiert. Nehmen Sie an, dass $\varepsilon = 1$ gilt. Erweitern Sie das Kuckuck-Hashing um eine geeignete `rehash`-Methode. Sie können zur Vereinfachung $\alpha_{\text{min}} = 1/16$, $\alpha_{\text{ideal}} = 1/8$ und $\alpha_{\text{max}} 1/4$ annehmen.
4. Testen Sie Ihr Programm für verschiedene Beispieleingaben. Fügen Sie Ihrer Abgabe zwei dieser Eingaben bei und erläutern Sie ausführlich, wie man das Programm kompilieren, starten und testen kann. Kommentieren Sie das Programm umfassend.

11-1

Kapitel 11
Entwurfsmethoden: Zufall

Gott würfelt nicht, Computer schon

11-2

Lernziele dieses Kapitels

1. Die Konzepte »zufällige Eingabe«, »zufällige Ausgabe« und »zufällige Entscheidung« unterscheiden können
2. Einschätzen können, wie verlässlich Zufallsalgorithmen sind
3. Wichtige Klassen von Zufallsalgorithmen erläutern können
4. Einfache Beispiele von Zufallsalgorithmen implementieren können

Inhalte dieses Kapitels

Worum es heute geht

Wenn man ehrlich ist, dann steht die Informatik mit dem Zufall ziemlich auf Kriegsfuß: Es wird *viel* technischer Aufwand getrieben, dass immer alles schön deterministisch abläuft und – in der Tat – wenn auch nur ein Bit im Code des Kernels eines Rechners mal spontan »umkippt«, dann stürzt der Rechner mit ziemlicher Sicherheit ab. Um so erstaunlicher ist es daher, dass der Zufall speziell in der Algorithmik eine ganz wichtige Rolle spielt. Hierfür gibt es zwei Gründe:

1. Zufallsalgorithmen sind oft viel schneller als »normale« Algorithmen…
2. …und dabei genauso zuverlässig.

Den ersten Punkt sieht man intuitiv noch recht leicht ein: Wenn ich die Antwort raten kann, dann geht das doch sicherlich schnell? Tatsächlich ist die Sachlage etwas komplizierter, da es *nicht* um Nichtdeterminismus geht, wo auf mehr oder weniger magische Weise Maschinen die richtigen Antworten auf ihren Bändern finden. Vielmehr müssen auch Zufallsalgorithmen »richtig hart arbeiten«, um die Lösung zu finden – der Zufalls »hilft« ihnen nur irgendwie dabei.

Beim zweiten Punkt wird aber sicherlich auch die wohlwollende Leserin skeptisch werden: Ist es nicht gerade das Wesen des Zufalls, dass »immer etwas schiefgehen kann«, wenn die Wahrscheinlichkeit auch gering sein mag? Und wie oft haben wir schon gehört, dass dieses oder jenes »so unwahrscheinlich ist, dass es praktisch ausgeschlossen ist«? (Als beispielsweise die ersten Atomkraftwerke in den 50er Jahren gebaut wurden, haben Professoren verkündet, dass eine Kernschmelze statistisch nur alle 27.000 Jahre vorkommen kann; also in der Praxis niemals. Nun ja.) Daher ist meiner Meinung nach die wichtigste Erkenntnis, die Sie bitte aus diesem Kapitel mitnehmen, folgende: Die Wahrscheinlichkeiten, dass gut

gebaute Zufallsalgorithmen »dummerweise« einen Fehler machen, ist so gering, dass dies *in diesem Universum nicht vorkommt.* Hier ein paar Dinge, die *in jeder Nanosekunde um Größenordnungen wahrscheinlicher* sind, als dass ein Zufallsalgorithmus »Pech hat«:

- Die Hardware des Rechners versagt spontan (ein Bit kippt um).
- Das Gebäude, in dem der Rechner steht, bricht spontan in sich zusammen.
- Die Stadt, in dem der Rechner steht, wird durch einen Asteroiden zerstört.
- Die Erde wird von einer Vogonischen Kriegsflotte zerstört, um einer Hyperraumumgehungsstraße Platz zu machen.

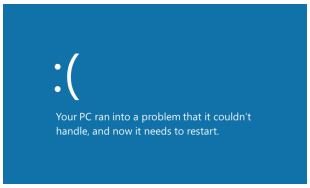
Man kann es auch so formulieren: Zufallsalgorithmen sind die größten Glückspilze, die sie sich vorstellen können. Sie haben einfach nie Pech.

11.1 Arten des Zufalls

Zufall in der Informatik – eine gute Idee?

11-4

Warum wir Zufall in der Informatik nicht mögen



- In der *Technischen* Informatik dürfen Bits im Speicher nicht umkippen.
- »Zufällige« Ergebnisse sind nicht reproduzierbar und lassen sich schlecht debuggen (»vorhin ging es noch«).

Warum wir Zufall in der Informatik trotzdem brauchen

1. Manche Eingaben sind wahrscheinlicher als andere. Algorithmen sollten das ausnutzen.
2. Kryptographische Algorithmen brauchen Zufallszahlen, da diese sich nicht erraten lassen.
3. »Zufallsalgorithmen« sind schneller als »normale« Algorithmen.

11.1.1 Zufällige Eingaben

Alle Eingaben sind gleich, aber manche sind gleicher als andere.

11-5

Eine Freundin sagt Ihnen: »Sortieren geht in Zeit $O(n \log n)$.« Damit meint sie, dass es einen Algorithmus gibt, der *jede Eingabe von n Zahlen* in Zeit $O(n \log n)$ sortieren kann. Bei manchen Eingaben kann es aber natürlich trotzdem schneller gehen (zum Beispiel, wenn die Eingabe schon sortiert ist).

Die Idee

Wenn »schnelle« Eingaben »häufig vorkommen in der Praxis«, dann ist die *erwartete Laufzeit* eventuell schneller als die Worst-Case-Laufzeit.

Man betrachtet dazu *Verteilungen* auf den möglichen Eingaben einer Wortlänge (zum Beispiel Gleichverteilungen). Dann betrachtet man den Erwartungswert der Laufzeit für diese Verteilung der Eingaben. Man nennt das Ergebnis die *Average-Case-Laufzeit* des Algorithmus.

Average-Case-Analysen sind mit Vorsicht zu genießen.

11-6

Leider ist es schwer, die »Verteilung der Eingaben in der Praxis« mathematisch zu beschreiben. Die Gleichverteilung ist es jedenfalls nicht, wie folgendes Beispiel zeigt:

Beispiel: Ein Algorithmus für das Erreichbarkeitsproblem

Folgender Algorithmus findet heraus, ab es einen Weg von s nach t in einem Graphen G gibt, indem er zunächst prüft, ob es einen Weg der Länge 2 gibt, und, wenn dies nicht der Fall ist, eine normale Tiefensuche durchführt.

```
1 input G = (V,E), s ∈ V, t ∈ V
2 foreach x ∈ V do
3   if (s,x) ∈ E ∧ (x,t) ∈ E then
4     return true
5 return solve_reachability_using_dfs(G,s,t)
```

▶ Satz
Die erwartete Laufzeit des Algorithmus ist konstant.

Beweis. Da alle Graphen gleich wahrscheinlich sind, gilt für alle Knoten $u, v \in V$, dass $\Pr[(u,v) \in E] = 1/2$. Damit gilt für jedes beliebige x, dass $\Pr[(s,x) \notin E \vee (x,t) \notin E] = 3/4$. Folglich wird Zeile 4 nach *genau k* Schleifendurchläufen erreicht mit Wahrscheinlichkeit $(3/4)^{k-1}(1/4)$. Wir *erwarten* deshalb, dass Zeile 4 erreicht wird nach Zeit

$$\sum_{k=1}^{n} k\left(\tfrac{3}{4}\right)^{k-1}\tfrac{1}{4} < 3.$$

Wird schließlich Zeile 4 nie erreicht, so benötigt Zeile 5 Zeit $O(n^2)$. Dies ist aber sehr unwahrscheinlich (nämlich $(3/4)^n$) und somit ist der Beitrag zur erwarteten Laufzeit von Zeile 5 nur $n^2(3/4)^n = o(1)$. □

11-7

Zusammenfassung zu zufälligen Eingaben.

Merke
Bei einer Average-Case-Analyse steckt der Zufall *in den Eingaben und in der Analyse*. Es ist in der Regel *unklar,* was die *»richtige«* Eingabeverteilung ist. Der Algorithmus *ist völlig deterministisch und nutzt keinen Zufall.*

11.1.2 Zufällige Ausgaben

11-8

Zufällige Ausgaben braucht man bei Simulationen und in der Statistik.

Will man reale Systeme *simulieren,* so braucht man oft viele *Zufallszahlen*. Das trifft auf *Computerspiele* genauso zu wie auf *Stichprobenanalysen*.

Die Idee
Ein *Pseudozufallszahlengenerator* ist eine leicht berechenbare Folge von Zahlen, die *»typische statistische Eigenschaften einer zufälligen Folge hat«.*

```c
static unsigned long int next = 1;

int rand(void) {
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

void srand(unsigned int seed) {
    next = seed;
}
```

11-9

Autor Sean MacEntee, Creative Commons License 2.0

Zufällige Ausgaben braucht man in der Kryptographie.

In der Kryptographie werden sehr oft *Strings oder Zahlen* benötigt, die man *nicht erraten kann*. Beispielsweise wird jede https-Verbindung mit einem (neuen) *Session-Key* verschlüsselt.

Die Idee
Für die Kryptographie braucht man Algorithmen, deren Ausgaben *»echte« Zufallszahlen* sind, die nicht vorhergesagt werden können.

Es ist eine *ganz schlechte Idee, Funktionen wie* `rand()` *in der Kryptographie zu nutzen* (warum?). »Echte« Zufallszahlen erzeugt man mittels *chaotischer physikalischer Vorgänge*, wie die Messung der Parität der Anzahl der Nanosekunden zwischen zwei Tastaturanschlägen.

11-10

By www.cgpgrey.com, CC BY 2.0

Zusammenfassung zu zufälligen Ausgaben.

Merke
In der Statistik und der Kryptographie braucht man Algorithmen, deren *Ausgabe möglichst zufällig ist*. Solche Algorithmen nennt man *Zufallszahlengeneratoren*. Für Anwendungen in der Statistik müssen diese möglichst einfach und *reproduzierbar* sein, in der Kryptographie *hingegen gerade nicht.*

11.1.3 Zufällige Entscheidungen

Algorithmen können »intern« zufällig arbeiten. 11-11

Algorithmen können *intern* den Zufall nutzen. Ein typisches Beispiel sind *Hash-Tabellen: Intern* werden die Elemente entsprechend der Hash-Funktionen »geschickt« verteilt, *»nach außen«* merkt man davon aber nichts, da die Grundoperationen *Suchen, Einfügen, Löschen* die *immer gleichen Ergebnisse liefern;* lediglich die *Geschwindigkeit* ändert sich. Überraschenderweise lässt sich Zufall aber *auch nutzen,* wenn lediglich »Ja/Nein«-Entscheidungen gefällt werden müssen wie bei der Frage »Ist x prim?«.

Die Idee

Ein *Algorithmus mit internem Zufall* soll *für alle Eingaben* (also im Worst-Case) *immer* das richtige Ergebnis bestimmen – es darf nur »mal schneller und mal langsamer« gehen in Abhängigkeit von zufälligen internen Entscheidungen.

Vergleich der Arten des Zufalls 11-12

Welche Aspekte sind vom Zufall beeinflusst?

Modell	Eingabe	Ausgabe	Berechnung	Laufzeit
Average-Case-Analyse	ja	nein	nein	nein
Pseudozufallszahlen	nein	pseudo	nein	nein
Kryptographische Schlüssel	nein	ja	nein	nein
Zufallsalgorithmen (ZPP)	nein	nein	ja	ja
Zufallsalgorithmen (BPP, RP)	nein	minimal	ja	nein

11.2 Zufallsalgorithmen: Beispiele

11.2.1 Termäquivalenz prüfen

Sind zwei Terme gleich? 11-13

Das Term-Äquivalenz-Problem

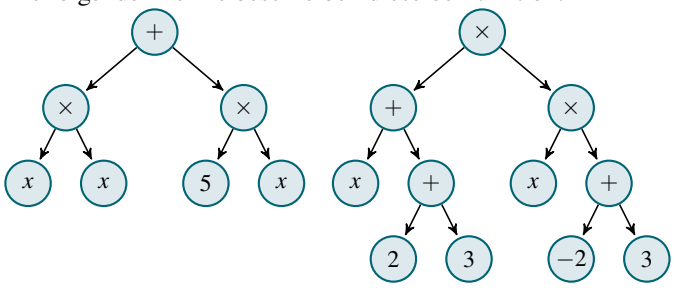
Eingabe Zwei arithmetische Terme s und t mit Variablen, kodiert als arithmetische Bäume mit Zahlen und Variablen an den Blätttern und $+$- und $\times$-Knoten im Inneren.

Frage? Beschreiben beide Terme dieselbe Funktion?

(Beachte: Die Eingabe ist syntaktisch, die Frage ist semantisch.)

Beispiel

Die folgenden Terme beschreiben dieselbe Funktion:



✎ **Zur Diskussion** 11-14

1. Wie schnell können Sie zwei Terme auf Äquivalenz überprüfen, wenn in ihnen *keine Variablen* vorkommen?
2. Wie schnell geht dies, wenn *genau eine Variable* vorkommt?

11-15

Die Ideen zur Lösung des Term-Äquivalenz-Problems

Von der Gleichheit zum Null-Test

Die arithmetischen Bäume beschreiben *Polynomfunktionen*. Nun sind zwei Polynomfunktionen, wenn ihre Differenz 0 ist. Wir müssen also »nur« herausfinden, ob die *durch einen arithmetischen Baum beschriebene Polynomfunktion die Nullfunktion ist*.

Big Idea

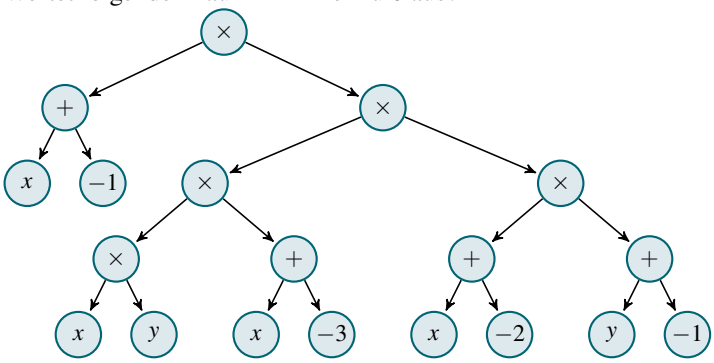
Polynomfunktionen, die nicht konstant 0 *sind, haben nur wenige Nullstellen!*

```
1 input tree T(x_1,...,x_k), where x_1,...,x_k are the variables
2 M ← geeignete Menge von Zahlentupeln // Siehe Übungen
3 foreach (a_1,...,a_n) ∈ M do
4   v ← T(a_1,...,a_k) // evaluate T for these numbers
5   if v ≠ 0 then
6     return ''T is not always 0''
7 return ''T is always 0''
```

11-16

Beispiel der Überprüfung

Wertet folgender Baum T immer zu 0 aus?



- Setzen wir $x = 0$ und $y = 0$, so ist $T(0,0) = 0$.
- Ebenso für alle anderen x und $y = 0$.
- Ebenso für alle x und $y = 1$.
- Ebenso für $x \in \{0,1,2,3\}$ und $y = 2$.
- *Aber* für $x = 4$ und $y = 2$ gilt $T(4,2) = 48$!

11-17

✎ Zur Übung

Wie viele Nullstellen haben folgende Polynom vom Grad d?

$$\begin{aligned} p_1(x,y,z) = {} & (x-1)(x-2)\cdots(x-d)+ \\ & (y-1)(y-2)\cdots(y-d)+ \\ & (z-1)(z-2)\cdots(z-d), \end{aligned}$$

$$p_2(x,y,z) = xyz.$$

11-18

Nullstellen trifft man nicht mit Dartpfeilen.

Ist $p(x_1,\ldots,x_n)$ nicht das Nullpolynom, so haben gerade gesehen, dass p zwar unendlich viele Nullstellen haben kann, aber:

- Sie werfen einen Dartpfeil auf den Raum $\mathbb{R}^n$.
- Dann ist die Wahrscheinlichkeit, dass der Pfeil eine Nullstelle trifft, gleich 0 (!), da die Nullstellen eine $(n-1)$-dimensionale (Hyper)fläche bilden.

Etwas mathematischer ausgedrückt:

$$\Pr_{(a_1,\ldots,a_n)\in\mathbb{R}^n}[p(a_1,\ldots,a_n) = 0] = 0.$$

Eine frustrierende Situation

- Im Raum $\mathbb{R}^n$ sind die Nullstellen von p so rar, dass sie »unmöglich zufällig zu treffen« sind.
- Wenn wir aber deterministisch das Gitter $\{1,\dots,d\}^n$ durchprobieren, können gerade dort überall Nullstellen sein.

11.2.2 Das Schwarz-Zippel-DeMillo-Lipton-Lemma

Wie wahrscheinlich ist es, eine Nullstelle zu treffen?

11-19

▶ Lemma: DeMillo, Lipton, Schwarz, Zippel
Sei $p(x_1,\dots,x_n)$ ein Polynom vom Grad d, das nicht konstant 0 ist, und sei $S \subseteq \mathbb{R}$ eine endliche Menge von Werten. Dann ist

$$\Pr_{(a_1,\dots,a_n)\in S^n}[p(a_1,\dots,a_n)=0] \le d/|S|.$$

Beweis. Für $n=1$ folgt das Lemma sofort daraus, dass $p(x)$ höchstens d Nullstellen haben kann. Für den Induktionsschritt von $n-1$ auf n schreiben wir p wie folgt um:

$$p(x_1,\dots,x_n) = \sum_{i=0}^{d} x_1^i q_i(x_2,\dots,x_n).$$

Da p nicht das Nullpolynom ist, gibt es ein maximales i, so dass q_i nicht das Nullpolynom ist. Sei j dieses i. Dann hat q_j höchstens Grad $d-j$, da das Monom $x_1^j q_j(\dots)$ höchstens Grad d hat.
Nun gilt für beliebige $(a_2,\dots,a_n) \in S^n$: Ist $q_j(a_1,\dots,a_n) \neq 0$, so ist $p(x_1,a_2,\dots,a_n)$ ein Polynom vom Grad j und somit ist $\Pr_{a_1\in S}[p(x_1,a_2,\dots,a_n)=0] \le j/|S|$. Damit ist die Gesamtwahrscheinlichkeit für $p(a_1,\dots,a_n)=0$ die Summe folgender Fälle:

1. $q_j(a_1,\dots,a_n)=0$, was nach Induktionsvoraussetzung mit Wahrscheinlichkeit $(d-j)/|S|$ passiert, und
2. $q_j(a_1,\dots,a_n)\neq 0$ und dann aber $p(x_1,a_2,\dots,a_n)=0$, was wie eben gezeigt dann mit Wahrscheinlichkeit $j/|S|$ passiert.

In der Summe ist die Wahrscheinlichkeit $(d-j)/|S| + j/|S| = d/|S|$. □

Ein Zufallsalgorithmus für den Term-Äquivalenz-Test.

11-20

```
1  input tree T(x_1,...,x_k), where x_1,...,x_k are the variables
2
3  foreach i ∈ {1,...,n} do
4    a_i ← pick_randomly_from({1,...,100d})
5
6  if T(a_1,...,a_k) ≠ 0
7    then return ''T is not always 0''
8    else return ''T is presumably always 0''
```

▶ Satz
Der Zufallsalgorithmus hat Laufzeit $O(n^2)$. Gibt er "T is not always 0" aus, so ist dies immer korrekt. Gibt er "T is presumably always 0" aus, so ist dies mit mindestens Wahrscheinlichkeit 99% korrekt.

Der Satz folgt unmittelbar aus dem Lemma.

11.2.3 Perfekte Matchings prüfen

11-21

Eine Anwendung: Perfekte Matchings finden.

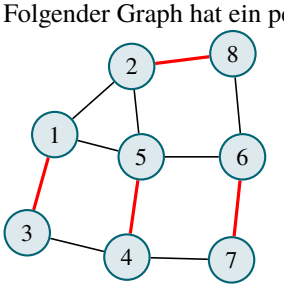
Das Perfekte-Matching-Problem

Eingabe Ein ungerichteter Graph.

Frage Enthält der Graph ein perfektes Matching? (Eine Teilmenge der Kanten, so dass jeder Knoten an genau einer Kante beteiligt ist?)

Beispiel

Folgender Graph hat ein perfektes Matching (rot dargestellt):



Man kann das Perfekte-Matching-Problem in polynomieller Zeit lösen, aber das ist sehr komplex. Wir werden jetzt einen Zufallsalgorithmus bauen, der viel einfacher und schneller ist.

11-22

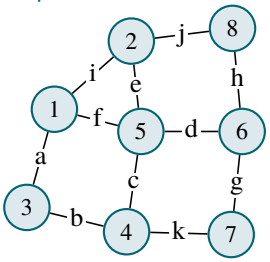
Die Tutte-Matrix

▶ **Definition**

Sei $G = (\{1,\dots,n\}, E)$ ein ungerichteter Graph. Die *Tutte-Matrix* T^G *von* G ist wie folgt definiert (die $v_{i,j}$ sind jeweils neue Variablen):

$$T^G_{ij} = \begin{cases} v_{i,j} & \text{für } (i,j) \in E, i < j, \\ -v_{j,i} & \text{für } (i,j) \in E, i > j, \\ 0 & \text{sonst.} \end{cases}$$

Beispiel



$$\begin{pmatrix} 0 & i & a & 0 & f & 0 & 0 & 0 \\ -i & 0 & 0 & 0 & e & 0 & 0 & j \\ -a & 0 & 0 & b & 0 & 0 & 0 & 0 \\ 0 & 0 & -b & 0 & c & 0 & k & 0 \\ -f & -e & 0 & -c & 0 & d & 0 & 0 \\ 0 & 0 & 0 & 0 & -d & 0 & g & h \\ 0 & 0 & 0 & -k & 0 & -g & 0 & 0 \\ 0 & -j & 0 & 0 & 0 & -h & 0 & 0 \end{pmatrix}$$

11-23

Die Tutte-Matrix und perfekte Matchings

▶ **Satz**

Ein Graph G hat genau dann ein perfektes Matching, wenn die Determinante von T^G *nicht das Nullpolynom ist.*

Beweis. Nach der Leibniz-Formel gilt:

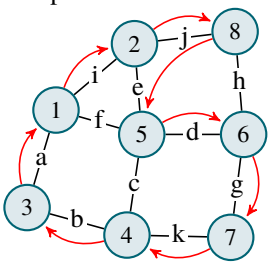
$$\det T^G = \sum_{\pi} (-1)^{\operatorname{sgn}\pi} T^G_{1,\pi(1)} T^G_{2,\pi(2)} \cdots T^G_{n,\pi(n)}.$$

Erste Beobachtungen:

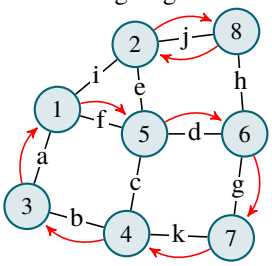
1. Jede Permutation π kann man eindeutig darstellen als Menge von *zyklischen Vertauschungen*.

2. Damit entspricht π einer *Überdeckung aller Knoten mit Hilfe von disjunkten Zyklen*.
3. Geht eine Kante entlang eines Zyklus *nicht* entlang einer Kante im Graphen, so ist der Beitrag der Permutation in der obigen Summe 0.
4. Es tragen also *genau die Überdeckungen von G mit Zyklen zu* $\det T^G$ *bei.*

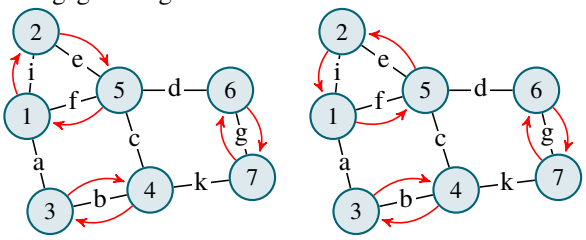
Beispielsweise ist der Beitrag folgender Permutation 0:



Der Beitrag folgender Permutation ist $j^2afdgkb$:



Nehmen wir für die erste Richtung an, G hat ein perfektes Matching $M = \{e_1, \ldots, en/2\}$. Dann gibt es genau eine Permutation π, die immer genau die beiden Knoten jeder Kante in M vertauscht. Diese Permutation trägt das Monom $e_1^2 e_2^2 \cdots en/2^2$ zum Polynom der Determinante bei, die somit nicht konstant 0 ist.
Für die zweite Richtung habe nun G kein perfektes Matching. Sei nun eine beliebige Permutation π gegeben, deren Zyklen »entlang von Kanten« verlaufen. Dann muss einer der Zyklen ungerade Länge haben, denn sonst betrachte jede zweite Kanten auf den Zyklen: Diese würden ein perfektes Matching bilden. Betrachte nun die Permutation π', die identisch ist zu π, aber *genau den ungerade Zyklus andersherum durchläuft*. Dann tragen π und π' *dasselbe Monom* zur Determinante bei, *aber mit umgekehrten Vorzeichen*, weshalb sie sich gegenseitig aufheben.



□

Ein Zufallsalgorithmus für das Perfekte-Matching-Problem

11-24

```
1  input graph G = ({1,...,n}, E)
2  T ← empty n × n matrix
3  foreach (i, j) ∈ E with i < j do
4     r ← pick_randomly_from({1,...,100n})
5     T[i, j] ← r
6     T[j, i] ← −r
7  d ← det(T) // kann in Zeit O(n^3) berechnet werden
8  if d ≠ 0
9     then return ''G has a prefect matching''
10    else return ''G presumably has no perfect matching''
```

▶ Satz

Der Zufallsalgorithmus hat Laufzeit $O(n^3)$. Gibt er "G has a prefect matching" aus, so ist dies immer korrekt. Gibt er "G presumably has no perfect matching" aus, so ist dies mit mindestens Wahrscheinlichkeit 99% korrekt.

Der Satz folgt aus dem Lemma und daraus, dass das Determinanten-Polynom Grad n ist.

11.3 Zufallsalgorithmen: Arten

11-25

Mögliche Arten von Fehlern bei Zufallsalgorithmen.

Am Anfang hieß es, ein Zufallsalgorithmus müsse *immer* das richtige Ergebnis ausgeben. Die vorgestellten Algorithmen machen aber *mit Wahrscheinlichkeit 1% einen Fehler, wenn sie »Nein« sagen*. Allgemein haben Zufallsalgorithmen fünf mögliche Ausgaben:

1. »Ja« und dies stimmt garantiert.
2. »Ja« und dies stimmt mit 99% Wahrscheinlichkeit.
3. »Unklar« und dies passiert mit 1% Wahrscheinlichkeit.
4. »Nein« und dies stimmt mit 99% Wahrscheinlichkeit.
5. »Nein« und dies stimmt garantiert.

▶ Definition: BPP-, RP-, coRP-, ZPP-Algorithmen

Ein *BPP-Algorithmus für ein Problem P* ist ein Algorithmus, der

1. in polynomieller Zeit arbeitet,
2. intern zufällige Entscheidungen fällen darf und
3. immer eine der fünf Ausgaben oben macht.

Spezialfälle von BPP-Algorithmen sind:

- RP-Algorithmen: Hier kommt Fall 2 nicht vor.
- coRP-Algorithmen: Hier kommt Fall 4 nicht vor.
- ZPP-Algorithmen: Hier kommen Fall 2 und 4 nicht vor.

11-26

BPP ist gut, RP und coRP sind besser, ZPP ist ideal

Merke

BPP	Die Ausgabe ist mit 99% Wahrscheinlichkeit richtig.
RP	Wie BPP, nur dass »Ja«-Ausgaben zu 100% richtig sind.
coRP	Wie BPP, nur dass »Nein«-Ausgaben zu 100% richtig sind.
ZPP	Die Ausgabe stimmt immer, aber mit 1% Wahrscheinlichkeit muss man die Berechnung wiederholen.

11.4 Zufällig und trotzdem zuverlässig?

11.4.1 Wahrscheinlichkeitsverstärkung

11-27

Kann man Zufallsalgorithmen vertrauen?

Aus *Health & Safety Executive Nuclear Directorate Assessment Report, New Reactor Build EDF/AREVA EPR Step 2 PSA Assessment* aus dem Jahr 2008:

> C[ore] D[amage] F[requency] ext[ernal] hazards: 7×10^{-8} extreme weather Results/y[ea]r

(Drei Jahre später trat ein ziemlich großer »Core-Damage« aufgrund von »extreme weather« in Fukushima ein.)

Unsere Zufallsalgorithmen haben eine 1% Chance, dass sie Fehler machen. »Gefühlt« ist das bei sicherheitskritischen Anwendungen »viel zu viel«. »Gefühlt« sollte man gar keine Algorithmen mit positiven Fehlerwahrscheinlichkeiten nutzen. *Man kann aber die Fehlerwahrscheinlichkeit verringern und die resultierenden Algorithmen sind dann »sicher«.*

11-28

Wie macht man Algorithmen sicher?

Gegeben sei ein RP-Algorithmus A, der ein Problem Q entscheidet (also bei Ausgabe »Ja« immer richtig liegt und bei Ausgabe »nein« zu 99% richtig liegt). Betrachte nun folgenden Algorithmus:

```
1 input x
2 do 500 times
3   a ← output of A on x
4   if a = »yes« then
5     return »yes«
6 return »no«
```

▶ Satz

Der obige Algorithmus ist ebenfalls ein RP-Algorithmus für A, aber mit Fehlerwahrscheinlichkeit 10^{-1000}.

11.4.2 Sehr unwahrscheinlich = nie

Eher stürzt ein Computer ab, als dass ein Ereignis mit Wahrscheinlichkeit 10^{-1000} eintritt. 11-29

Ein Speicherriegel hat (Messung aus dem Jahr 2009) in einem typischen realen Rechner mindestens alle 10 Jahre einen »unrecoverable error«. Damit ist die Wahrscheinlichkeit, dass ein Computer in einer gegebenen Nanosekunde aufgrund eines Speicherfehlers abstürzt, mindestens

$$\frac{1}{\underbrace{10^9}_{\text{Nanosekunden pro Sekunde}} \cdot \underbrace{10^9}_{\text{Sekunden pro 10 Jahre}} \cdot \underbrace{10^{12}}_{\text{Mögliche aktuelle Speicheradresse}}}$$

also mindestens 10^{-30}. Das ist *viel, viel mehr* als 10^{-1000}.
Aber, ist »sehr unwahrscheinlich« nicht »aber trotzdem möglich«?

Ein Ereignis mit Wahrscheinlichkeit 10^{-1000} tritt in diesem Universum nicht ein. 11-30

Betrachten wir die Wahrscheinlichkeit, dass eine Ereignis eintritt

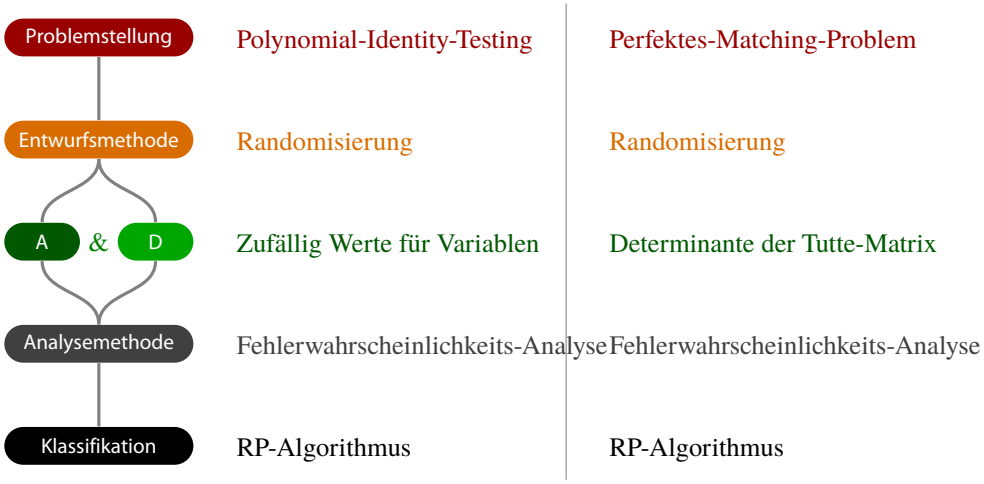
- bei irgendeinem Atom im Universum (10^{80})
- während einer der Planck-Zeiten innerhalb der ersten Sekunde nach dem Urknall (10^{45})
- oder während einer Planck-Zeit innerhalb aller Sekunden seit dem Urknall (10^{18})
- oder während einer Milliarde weiterer Zeitalter des Universums (10^9).

Es gibt also $10^{80+45+18+9} = 10^{152}$ Möglichkeiten, zu denen das Ereignis eintreten kann. Wenn dies jedes Mal eine Chance von 10^{-1000} hatte, so passiert dies wenigstens einmal mit Wahrscheinlichkeit

$$10^{-848}.$$

Also nie.

Zusammenfassung dieses Kapitels



11-31

▶ **Zufallsalgorithmus**

- Ein *Zufallsalgorithmus* nutzt *intern* Zufallsentscheidungen.
- Ob der Algorithmus schnell arbeitet oder das richtige Ergebnis liefert, hängt *nicht* von der Eingabe ab.
- Man kann Zufallsalgorithmen *wiederholt* auf eine Eingabe anwenden, um die *Fehlerwahrscheinlichkeit beliebig klein zu bekommen*.
- Zufallsalgorithmen machen *in der Praxis nie (!) Fehler*.

Übungen zu diesem Kapitel

Übung 11.1 Nullstelen eines Polynom auf einem Gitter, mittel

Sei $p(x_1,\ldots,x_n)$ ein Polynom vom Grad d ungleich dem Nullpolynom. Zeigen Sie, dass für wenigstens ein Tupel $(a_1,\ldots,a_n) \in \{1,\ldots,d+1\}$ gilt $p(a_1,\ldots,a_n) \neq 0$.

Tipp: Sie können dies entweder direkt durch Induktion über n beweisen oder das Schwarz-Zippel-DeMillo-Lipton-Lemma benutzen (nur wie?...).

Übung 11.2 Algorithmus für den Term-Nullpolynom-Test, leicht

In dem Algorithmus von Folie **??** wählen Sie $M = \{1,\ldots,d+1\}^n$. Zeigen Sie:

1. Der Algorithmus liefert nun immer das richtige Ergebnis. (*Tipp:* Nutzen Sie Übung 11.1.)
2. Die Laufzeit des Algorithmus ist beschränkt durch $O(n^n)$.

Übung 11.3 BPP-Algorithmen sicher machen, schwer

Sie haben einen BPP-Algorithmus A gegeben, der ein Problem Q löst, der also in polynomieller Zeit zu einer Eingabe x die Ausgabe »$x \in Q$« oder »$x \notin Q$« macht und dies stimmt mit 99% Wahrscheinlichkeit.

Die Fehlerwahrscheinlichkeit des Algorithmus liegt also unter 2^{-6}. Sie suchen nun einen Algorithmus, dess Fehlerwahrscheinlichkeit unter 2^{-1000} liegt.

Wie können Sie A modifizieren, um dies zu erreichen? Geben Sie den Algorithmus und seine Wahrscheinlichkeitsanalyse an.

Tipp: Chernoff-Ungleichung

Teil V

Entwurfsmethoden für schwere Probleme

Die in den vorherigen Kapiteln untersuchten Problemen waren einfach.

Wer sich durch die Details des DC3-Algorithmus geackert hat oder wer versucht hat, die amortisierte Analyse des Kuckucks-Hashing nachzuvollziehen, der mag den vorherigen Satz für arrogant halten, er stimmt aber trotzdem: Im Sinne der *Komplexitätstheorie* von Problemen ist das Erstellen eines Suffix-Arrays ein fast triviales Problem, geht es doch sicherlich in polynomieller Zeit zu lösen. Aus Sicht der klassischen Komplexitätstheorie ist alles »leicht«, was in der Klasse P liegt; die Schwierigkeiten fangen erst oberhalb davon an. Man könnte argumentieren, dass die komplexitätstheoretische Sicht auf Probleme Theoretiker erfreuen mag, sie doch aber eine falsche Sicht zumindest auf das Algorithmendesign ist: Es macht praktisch eben doch einen himmelweiten Unterschied, ob man einen Suffix-Array in Zeit $O(n)$ oder in Zeit $\Theta(n^2 \log n)$ berechnen kann.

Ziel dieses letzten Teils ist zu zeigen, dass die Untersuchung der Komplexität von Problemen sehr wohl dazu beitragen kann, sie effizient zu lösen. Bekanntermaßen gibt es viele NP-vollständige Probleme, die sich nach heutigem Kenntnisstand *nicht* in polynomieller Zeit lösen lassen. Bei solchen Problemen *muss* man zu anderen Methoden greifen als zu denen, die wir für die »einfachen« Probleme aus den vorherigen Kapiteln kennengelernt haben. Besonders wichtig ist dabei, zunächst überhaupt erst zu *erkennen*, dass ein Problem schwierig ist – wie wir sehen werden, steckt der Teufel hier oft im Detail, schon eine leichte Umformulierung kann die Komplexität eines Problems drastisch ändern.

Wenn nun aber ein Problem als zweifelsfrei schwierig zu lösen identifiziert wurde, was dann? Wir werden uns zwei Verfahren etwas genauer anschauen: Zum einen Approximationsalgorithmen, welche in vielen Fällen sehr gute Ergebnisse liefern, und zum anderen Fixed-Parameter-Verfahren. Letztere funktionieren zwar nur unter »günstigen Bedingungen«, liefern dann aber spektakuläre Ergebnisse: Probleme, die sich auf Supercomputern nur bis Eingabegrößen von vielleicht $n = 50$ lösen ließen konnten hiermit auf normalen Computern für $n = 500$ gelöst werden.

12-1

Kapitel 12
Entwurfsmethode: Approximation
Die 80-20-Regel

12-2

Lernziele dieses Kapitels

1. Die Entwurfsmethode »Approximation« kennen und Approximationsalgorithmen entwerfen können
2. Wichtige Approximationsalgorithmen für mehrere schwere Probleme kennen

Inhalte dieses Kapitels

Worum es heute geht

Für viele der schweren Probleme, die wir in den letzten Kapiteln kennen gelernt haben, ist es, mit Verlaub gesagt, bescheuert, die Probleme exakt zu lösen. Nehmen wir dazu das Beispiel des Handelsreisenden. Wie wir gesehen haben, ist dieses Problem NP-vollständig, weshalb es bei großen Eingaben sehr lange dauert oder schlichtweg unmöglich ist, optimale Rundreisen zu berechnen. Bei realen Eingaben werden aber beispielsweise die Entfernungen, die ja Teil der Eingabe sind, mit einem gewissen Fehler behaftet sein; die kürzeste Rundreise für diese fehlerbehafteten Eingaben muss gar nicht die real kürzeste Rundreise sein. Was wir *eigentlich* suchen, sind Lösungen für das Handelsreisenden-Problem, die »dicht dran« sind am Optimum. Wenn die Eingaben beispielsweise eine Messungenauigkeit von, sagen wir, einem Prozent haben, so reicht es, eine Lösung zu finden, die bis zu einem Prozent vom Optimum abweichen darf – genauere Lösungen sind reine Augenwischerei.

Ist es leichter, solch »approximative« Lösungen für NP-vollständige Probleme zu berechnen, als diese exakt zu lösen? Hier kann man nur die »Juristen-Antwort« geben: Es kommt drauf an. Bei einigen NP-vollständigen Problemen ist es *viel* leichter, diese approximativ zu lösen, bei anderen ist schon dies schwierig. Die *Approximationstheorie* ist ein (sehr weit entwickeltes) Teilgebiet der Theorie, das sich genau mit der Frage beschäftigt, welche Probleme sich gut approximieren lassen und welche nicht.

In diesem Kapitel werden wir eine ganze Reihe von Approximationsalgorithmen kennenlernen. Dies soll aber nicht darüber hinwegtäuschen, dass es solche Algorithmen eben nicht immer gibt. Dies ist beispielsweise für das Färbeproblem der Fall: Man kennt keinen effizienten Algorithmus, um beliebige Graphen mit einer minimalen Anzahl an Farben zu färben (so dass benachbarte Knoten unterschiedliche Knoten haben); man kennt noch nicht einmal einen Algorithmus, der einen beliebigen Graphen mit, sagen wir, einer Million mal so vielen Farben wie unbedingt nötigt färbt. Wie immer in der Theorie sind allerdings solche »unteren Schranken« schwierig zu beweisen, weshalb hierfür auf weiterführende Veranstaltungen verwiesen sei.

12.1 Optimierungsprobleme

12.1.1 Das Konzept

Worum geht es bei »Problemen« wirklich?

12-4

Bei vielen Problemen gibt es nicht »die eine richtige Antwort«. Vielmehr gibt es »viele richtige Lösungen«, auch wenn diese »unterschiedlich gut« sind. Unsere bisherige Formalisierung von Problemen als einfache Ja/Nein-Probleme wird diesem Umstand nicht gerecht.

▶ Definition: Optimierungsproblem

Ein *Optimierungsproblem* für ein Alphabet Σ besteht aus:

1. Einer Lösungsrelation S. Dies ist eine Teilmenge von $\Sigma^* \times \Sigma^*$. Ist ein Paar (i,s) in dieser Menge, so heißt *s eine Lösung zu der Eingabe i*.
2. Einer Maßfunktion. Dies ist eine Funktion $m\colon S \to \mathbb{N}$, die jeder Lösung ein Maß zuordnet.
3. Einem Typ. Dieser ist entweder »Maximierung« oder »Minimierung«.

Beispiel: Vertex-Cover als Optimierungsproblem

12-5

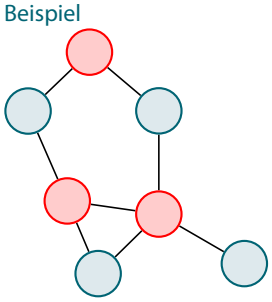
Eingaben Ungerichtete Graphen $G = (V,E)$.

Lösungen Kontenmengen M, so dass für jede Kante von G mindestens eines ihrer Enden in M liegt. (Für alle $\{u,v\} \in E$ muss gelten $u \in M$ oder $v \in M$.)

Maß Größe von M.

Ziel Minimierung.

Beispiel



✎ Zur Übung

Formulieren Sie das Handelsreisenden-Problem sinnvoll als formales Optimierungsproblem.

12.1.2 Maß und Güte

Gute versus schlechte Lösungen.

12-6

▶ Definition

Sei P ein Optimierungsproblem. Dann ist

- das *optimale Maß* zur Eingabe das minimale (oder maximale) Maß aller Lösungen zu x,
- die *Güte* einer konkreten Lösung s zu x das Verhältnis

$$\frac{\text{Maß von } s}{\text{optimales Maß einer Lösung zu } x}.$$

Bei Maximierungsproblemen ist die Güte gerade der Kehrwert, so dass sie immer mindestens 1 ist.

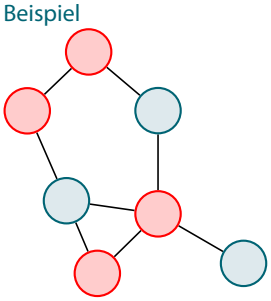
Merke

Das *Maß* einer Lösung gibt ihre *»absoluten Kosten«* an. Die *Güte* einer Lösung gibt an, *um welchen Faktor* sie schlechter ist als eine optimale Lösung.

12-7

Beispiele für Maß und Güte.

Beispiel



1. Das Maß der roten Lösung ist 4 (vier Knoten).
2. Das Maß der *optimalen* Lösung (drei Knoten) ist 3.
3. Damit ist die *Güte* der roten Lösung $4/3$.

12.2 Approximationsalgorithmen

12.2.1 Das Konzept

12-8

Die 80-20-Regel.

Eine Lebensweisheit

Die 80-20-Regel ist eine *empirische Beobachtung* betreffend viele reale Projekte:

- Es werden 80% der Arbeit in 20% der Zeit erledigt.
- Umgekehrt benötigen 20% der Arbeit 80% der Zeit.

Beispiel

Wenn Sie eine Bachelor-Arbeit schreiben, werden 80% des Textes flott von der Hand gehen. Sie werden dann aber 80% Ihrer Zeit auf den »Feinschliff« des Textes verwenden.

Beispiel

Wenn Sie ein Programm schreiben, werden 80% des Codes schnell geschrieben sein. Sie werden dann aber 80% Ihrer Zeit auf den »Feinschliff« des Codes verwenden.

Big Idea

Lasse den Feinschliff weg! (Und spare 80% der Zeit.)

12-9

Formale Definition von Approximationalgorithmen.

Definition

Sei P ein Optimierungsproblem und $r \geq 1$. Ein *r-Approximationsalgorithmus für P* ist ein (Polynomialzeit-)Algorithmus, der für jede Eingabe x eine Ausgabe s mit folgenden Eigenschaften liefert:

1. s ist eine Lösung für x, das heißt, $(x, s) \in S$.
2. Die Güte von s ist kleiner oder gleich r.

Beispiel

Ein 2-Approximationsalgorithmus für das Handelsreisenden-Problem liefert immer eine Rundreise, die *höchstens doppelt so lang* wie kürzeste Rundreise ist.

Beispiel

Ein 1,01-Approximationsalgorithmus für das Rucksack-Problem liefert immer eine Auswahl von Gegenständen, deren Wert *bis auf 1%* an den Wert der optimalen Auswahl herankommt.

12.2.2 Handelsreisender in der Ebene

Der Handelsreisende in der Ebene

12-10

▶ Definition: MIN-TSP

Eingaben Eine Menge M von *Städten* und eine Funktion $m\colon M \times M \to \mathbb{N}$, die je zwei Städten *Reisekosten* zwischen diesen Städten zuordnet.

Lösungen Rundreisen durch die Städte, so dass jede Stadt genau einmal besucht wird.

Maß Kosten der Rundreise.

Ziel Minimierung.

▶ Definition: Euklidisches Handelsreisendenproblem

Das Problem MIN-EUCLIDIAN-TSP ist definiert wie MIN-TSP, nur muss M eine Menge von *Punkten in der Ebene* sein und m muss Städtepaare auf ihre *Distanz in der Ebene* abbilden (gerundet).

▶ Satz

Die Entscheidungsvariante EUCLIDIAN-TSP *ist* NP*-vollständig.*

Beweisideen. Man führt folgende Kette von Reduktionen durch:

1. Reduziere HAMILTON auf PLANAR-HAMILTON (nur planare Graphen sind als Eingabe zugelassen), indem kreuzende Kanten mittels geeigneter Gadgets simuliert werden.
2. Reduziere nun weiter auf GRID-HAMILTON (nur Teilgraphen eines Gitters sind als Eingaben zugelassen). Hierzu ersetzt man Knoten durch kleine Quadrate und Kanten durch lange, zwei Knoten breite »Tentakel«.
3. Reduziere dann weiter auf EUCLIDIAN-TSP. □

Das Handelsreisendenproblem in der Ebene ist 2-approximierbar.

12-11

▶ Satz

Es gibt einen 2-Approximationsalgorithmus für MIN-EUCLIDIAN-TSP.

Beweis. Der Algorithmus funktioniert für eine Menge M von Städten so:

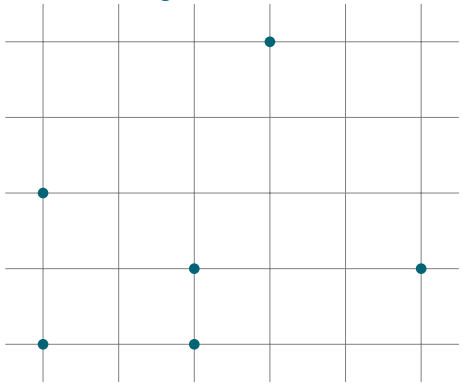
1. Erzeuge einen vollständigen Graphen, dessen Knoten Städte sind und dessen Kanten mit den Entfernungen der Städte gewichtet sind.
2. Berechne ein minimales Gerüst des Graphen.
3. Erzeuge eine Eulertour aus dem Gerüst durch *Verdopplung aller Kanten*.
4. Verkürze die Eulertour, bis jeder Knoten nur einmal besucht wird.

Dass dieser Algorithmus eine 2-Approximation liefert, sieht man so: Die kürzeste Rundreise habe die Länge x. Löschen wir daraus eine Kante, so erhalten wir einen Pfad, der noch kürzer ist. Dieser Pfad ist ein Gerüst. Also ist das Gewicht y des minimalen Gerüsts noch kleiner: $y < x$. Die Eulertour hat Länge $2y < 2x$. Die ausgegebene Tour ist kürzer als die Eulertour, also kürzer als $2x$. □

Der Algorithmus an einem Beispiel.

12-12

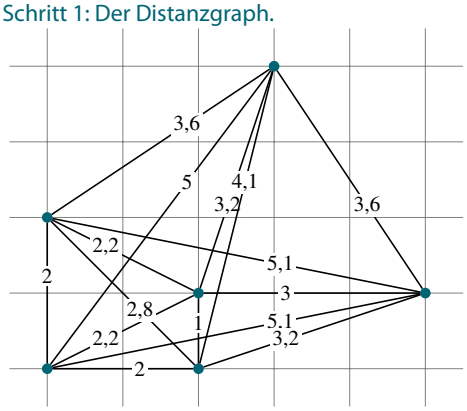
Schritt 0: Die Eingabe.



12-13

Der Algorithmus an einem Beispiel.

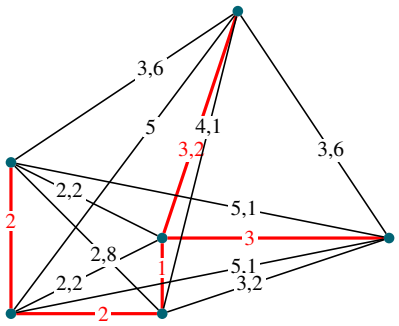
Schritt 1: Der Distanzgraph.



12-14

Der Algorithmus an einem Beispiel.

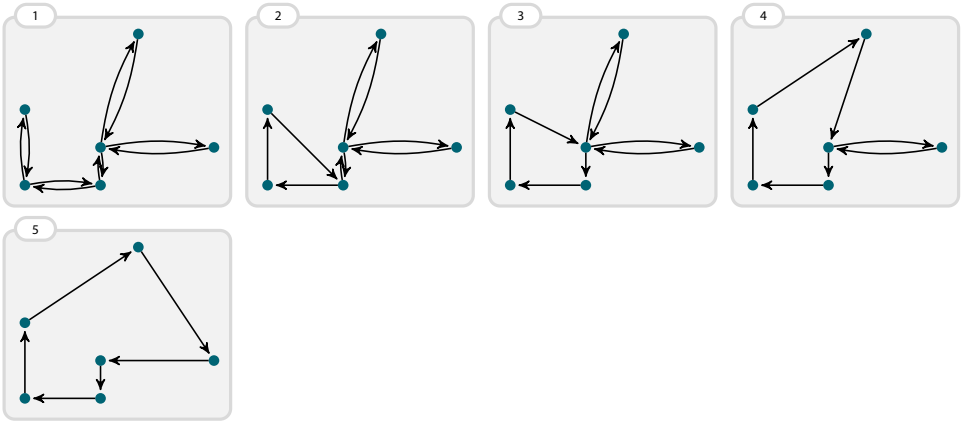
Schritt 2: Das Gerüst.



12-15

Der Algorithmus an einem Beispiel.

Schritte 3 und 4: Geradeziehen der Eulertour.



12-16

Das Handelsreisendenproblem in der Ebene ist 1,5-approximierbar.

▶ Satz: Christofides
Es gibt einen 1,5-Approximationsalgorithmus für MIN-EUCLIDIAN-TSP.

Beweis. Der Algorithmus ist ähnlich wie eben:

1. Erzeuge den vollständigen Graphen G.
2. Berechne ein minimales Gerüst R des Graphen.
3. *Betrachte die Menge U aller Knoten in R mit ungeradem Grad (in Bezug auf R).*
4. *Berechne ein Matching M mit minimalem Gewicht auf dem von U in G induzierten Teilgraphen.*
5. Vereinigt man R und M, so hat jeder Knoten geraden Grad und somit gibt es eine Eulertour.

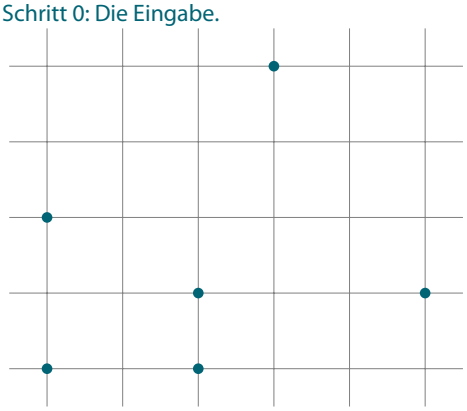
6. Verkürze die Eulertour, bis jeder Knoten nur einmal besucht wird.

Dieser Algorithmus liefert eine 1,5-Approximation: Die kürzeste Rundreise habe die Länge x. Wieder hat das Gerüst ein Gesamtgewicht von $y < x$. Weiter hat das Matching ein Gesamtgewicht von $z < x/2$, denn jede Rundreise durch den ganzen Graphen induziert eine (nur noch kürzere) Rundreise durch nur die Knoten von U, indem man Knoten außerhalb von U überspringt. Aus dieser Rundreise kann man zwei Matchings gewinnen, indem man nur die Kanten an *gerade* oder nur die an *ungerade* Stellen betrachtet. Das minimales Matching M ist folglich mindestens so gut wie das bessere dieser Matchings und hat deshalb *höchstens das halbe Gewicht der besten Rundreise durch G*. Damit hat die Eulertour Länge $y+z < \frac{3}{2}x$ und die ausgegebene Tour ist kürzer als die Eulertour, also kürzer als $\frac{3}{2}x$. □

Der Christofides-Algorithmus an einem Beispiel.

12-17

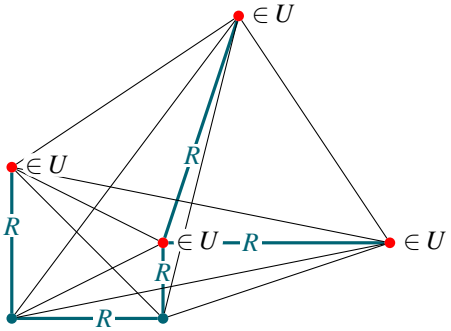
Schritt 0: Die Eingabe.



Der Christofides-Algorithmus an einem Beispiel.

12-18

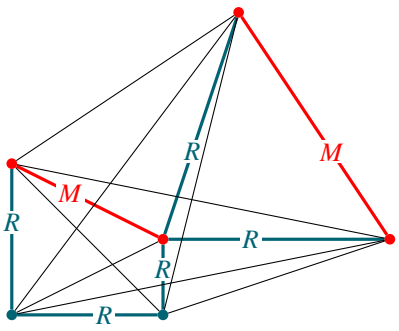
Schritt 1: Das Gerüst und die Menge U.



Der Christofides-Algorithmus an einem Beispiel.

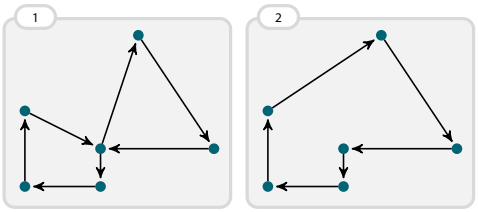
12-19

Schritt 2: Das Matching auf U in G.



12-20

Der Christofides-Algorithmus an einem Beispiel.
Schritt 3: Die Eulertour auf $R \cup M$ und das Geradeziehen.



12.2.3 Bin-Packing

12-21

Illustration des Bin-Packing Problems.

→

12-22

Das Bin-Packing-Problem

▶ Definition: MIN-BIN-PACKING

Eingabe Eine Liste von Objektgrößen $(g_1, \ldots, g_n)$ und eine Eimergröße b.

Lösungen Zuordnung von Objekten zu Eimern, so dass die Summe der Größen aller Objekte, die demselben Eimer zugeordnet sind, maximal b ist.

Maß Anzahl der benutzten Eimer.

Ziel Minimierung.

Man kann mittels einer Reduktion von TRIPARTITE-MATCHING zeigen:

▶ Satz
Die Entscheidungsvariante BIN-PACKING *ist* NP*-vollständig.*

12-23

Ein Greedy-Algorithmus für Bin-Packing.

▶ Satz
Es gibt einen 2-Approximationsalgorithmus für MIN-BIN-PACKING.

Beweis. Der Algorithmus wiederholt für jeden Gegenstand:

1. Finde, von links beginnend, den ersten Eimer, in den der Gegenstand noch passt.
2. Platziere den Gegenstand in diesen Eimer.

Dies funktioniert: Betrachten wir eine Lösung, die First-Fit produziert hat. Dann *passt der Inhalt von je zwei* nebeneinander stehenden Eimern *nicht* in einen einzigen Eimer (sonst hätte First-Fit das nämlich getan). Also muss auch in einer optimalen Lösung für je zwei von First-Fit benutzte Eimer mindestens ein Eimer genutzt werden. □

12.2.4 Vertex-Cover

Vertex-Cover lässt sich sehr leicht approximieren. 12-24

▶ Satz
Die Entscheidungsvariante VC *ist* NP*-vollständig.*

Beweis. Triviale Reduktion von INDEPENDENT-SET. □

▶ Satz
Es gibt einen 2-Approximationsalgorithmus für MIN-VC.

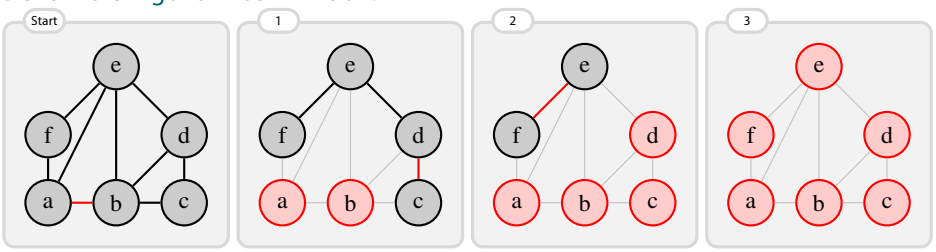
Beweis. Verwende folgenden Algorithmus:

```
1 while there is an uncovered edge do
2   pick any edge {u, v}
3   add both u and v to the vertex cover
4   delete u and v and all pending edges from the graph
```

Da von jeder Kante mindestens ein Endpunkt gewählt werden muss, ist die gefundene Lösung höchstens doppelt so groß wie die optimale. □

Es ist kein besserer Algorithmus bekannt.

Der triviale Algorithmus in Aktion. 12-25



Ein »offensichtlich besserer«, greedy-basierter Algorithmus. 12-26

Nehmen wir einen Knoten *nicht* nach M auf, so *müssen* wir *alle Nachbarn* aufnehmen. Daher erscheint es sinnvoll, *immer den Knoten mit höchstem Grad zu wählen*.

Greedy-Algorithmus für Vertex-Cover

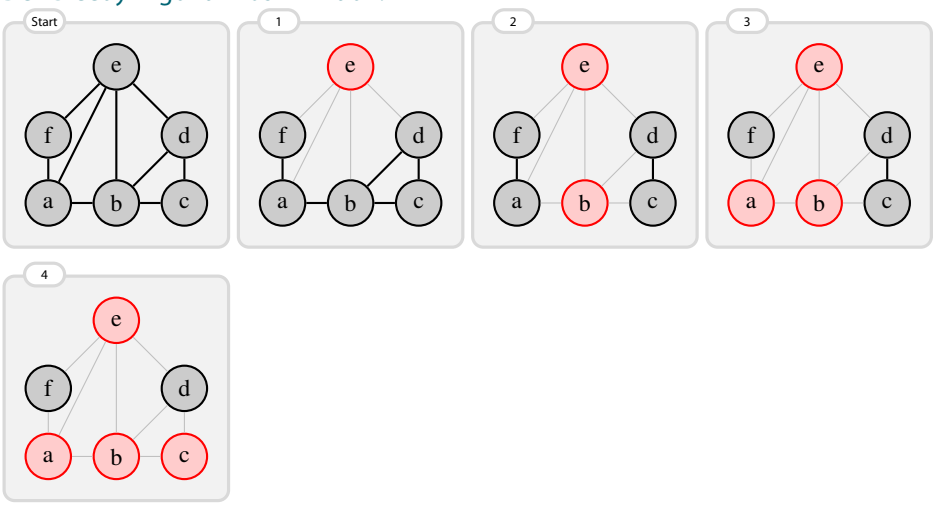
```
1 while there is an uncovered edge do
2   pick a vertex v of maximum degree
3   add v to the vertex cover
4   delete v and all pending edges from the graph
```

▶ Satz
Der Greedy-Algorithmus ist kein Approximationsalgorithmus.

Beweis. In Übung 12.1 wird gezeigt, dass die Approximationsgüte bis zu $\ln n$ betragen kann (und somit unbeschränkt ist). □

12-27

Der Greedy-Algorithmus in Aktion.



12.3 *Approximationsschemata

12.3.1 Das Konzept

12-28

Wie viel sollten Sie für eine Klausur lernen?

Bob hat folgende Beobachtung beim Lernen für seine Klausuren gemacht:

- Lernt er zwei Stunde lang, so besteht er.
- Lernt er vier Stunden lang, so bekommt er im Schnitt eine 3,0.
- Lernt er acht Stunden lang, so bekommt er im Schnitt eine 2,0.
- Lernt er 16 Stunden lang, so bekommt er im Schnitt eine 1,7.
- Lernt er 32 Stunden lang, so bekommt er im Schnitt eine 1,3.
- Lernt er 64 Stunden lang, so bekommt er im Schnitt eine 1,2.
- Lernt er 128 Stunden lang, so bekommt er im Schnitt eine 1,1.
- Selbst wenn er beliebig viel lernt, so kann er keine 1,0 garantieren.

✎ Zur Diskussion

Wie viel sollte Bob für eine Klausur lernen?

12-29

Die Idee des Approximationsschemas.

▶ Definition

Sei P ein Optimierungsproblem. Ein *Approximationsschema* ist eine *Familie von Algorithmen* A_k für $k \in \{1,2,3,\dots\}$, so dass A_k gerade ein $(1+1/k)$-Approximationsalgorithmus für P ist.

Man kann also P »beliebig gut approximieren«, jedoch kann dies »immer aufwändiger« werden mit steigendem k.

12.3.2 Rucksack-Problem

12-30

Zur Erinnerung: Das Rucksack-Problem

Copyright by Marieke Kuijer, Creative Commons Attribution-ShareAlike

→

Copyright by user Joadl, Creative Commons Attribution-ShareAlike

▶ Problem: Das Optimierungsproblem MAX-KNAPSACK

Eingabe Eine Folge $(w_1,\ldots,w_n)$ von *Werten* und eine Folge $(g_1,\ldots,g_n)$ von *Gewichten* und ein Maximalgewicht m.

Lösungen Teilmengen $I \subseteq \{1,\ldots,n\}$, so dass $\sum_{i\in I} g_i \leq m$.

Maß Der Gesamtwert $\sum_{i\in I} w_i$ der ausgewählten Gegenstände.

Ziel Maximierung.

Es gibt ein Approximationsschema für das Rucksack-Problem

12-31

▶ Satz

Es gibt ein Approximationsschema für das Rucksack-Problem.

Beweisplan.

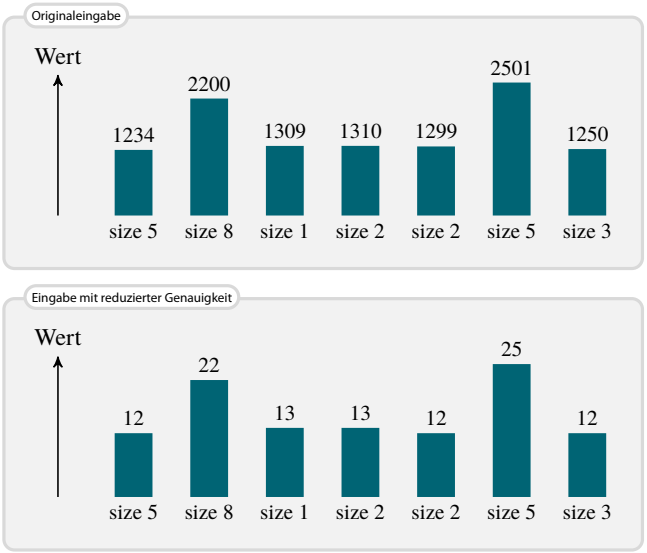
1. Zur Erinnerung: In Übung 2.2 haben wir einen Algorithmus kennen gelernt, der das Problem in Zeit $O(nW)$ löst, wobei W der Gesamtwert aller Gegenstände ist.
2. Für große W dauert dies zwar zu lange, aber *wenn wir das Problem nur approximativ lösen* wollen, so genügt es, die Werte *zu runden*.
3. Statt großer, genauer Werte wie $(120001, 320016, 420004, 150067)$ rechnet man mit den kleinen Werten $(12, 32, 42, 15)$, was dann schnell geht.
4. Wie stark man rundet, macht man von der gewünschten Genauigkeit abhängig. □

Skript: Beweis-details

Das Approximationsschema

```
1 input Werte (w_1,...,w_n)
2 input Größen (g_1,...,g_n)
3 input Rucksackgröße m
4 input k
5 μ ← max{w_1,...,w_n}
6 for i ← 1 to n do
7   w'_i ← ⌊w_i n(k+1)/μ⌋
8 return the set I computed by the exact algorithm on (w'_1,...,w'_n,g_1,...,g_n,m)
```

Ein Beispiel für die Rundung



Die Korrektheit des Tricks sieht man so: Sei J eine Lösung für die Originaleingabe und sei $x =$

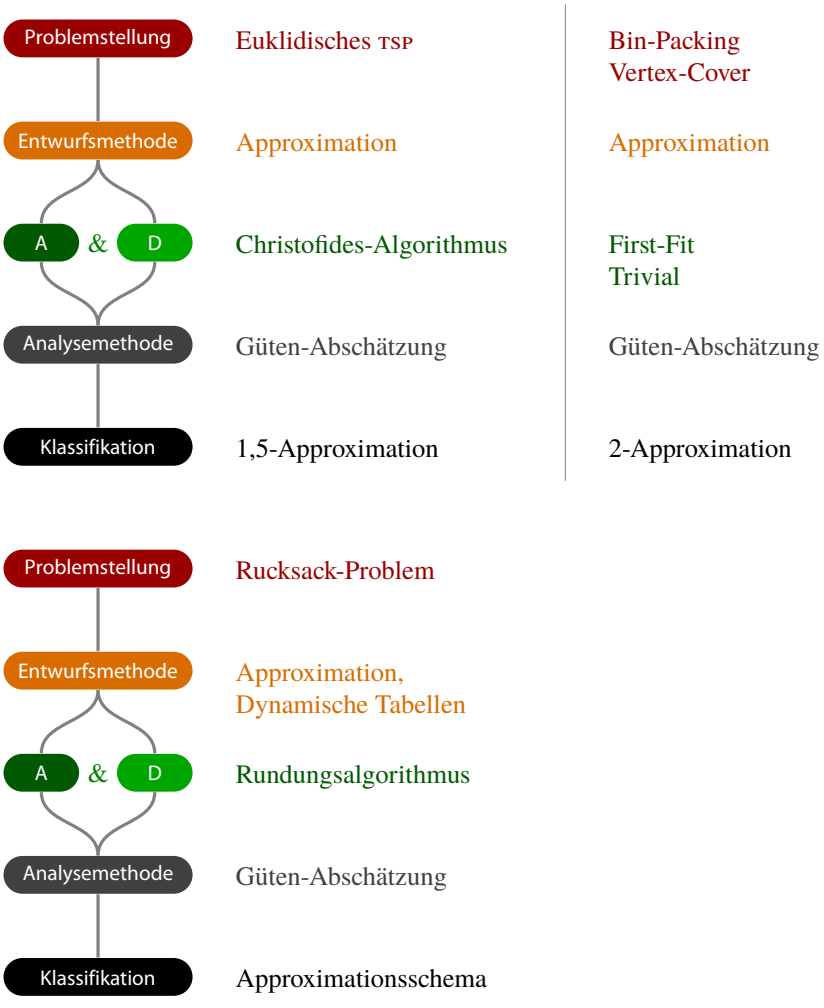
$\sum_{j\in J} w_j$ deren Wert. Sei weiter I die optimale Lösung für $(w'_1,\ldots,w'_n,g_1,\ldots,g_n,m)$. Dann gilt:

$$\begin{aligned}\sum_{i\in I} w_i &= \frac{\mu}{nk}\sum_{i\in I} w_i\frac{n(k+1)}{\mu} \geq \frac{\mu}{nk}\sum_{i\in I}\left\lfloor w_i\frac{n(k+1)}{\mu}\right\rfloor\\ &\geq \frac{\mu}{nk}\sum_{j\in J}\left\lfloor w_j\frac{n(k+1)}{\mu}\right\rfloor \geq \frac{\mu}{nk}\sum_{j\in J}(w_j\frac{n(k+1)}{\mu}-1)\\ &\geq x-\frac{n\mu}{n(k+1)}\\ &\geq x-x/(k+1) = x(1-1/(k+1)) = x/(1+1/k).\end{aligned}$$

Der Wert der gefundenen Lösung kommt also bis auf den Faktor $1+1/k$ an den Wert der optimalen Lösung heran.

Zusammenfassung dieses Kapitels

12-32



▶ Approximationsalgorithmus
Ein *r-Approximationsalgorithmus* für ein Optimierungsproblem P ist ein Polynomialzeitalgorithmus, der bei jeder Eingabe x eine Lösung der Güte kleiner oder gleich r liefert.

▶ Approximationsschema
Ein *Approximationsschema* für ein Optimierungsproblem P ist eine Folge A_1, A_2, A_3 und so weiter, so dass A_k ein $(1+1/k)$-Approximationsalgorithmen für P ist.

▶ NP-Vollständigkeit ist nicht dasselbe wie Approximierbarkeit
Für viele NP-vollständige Optimierungsprobleme kennt man Approximationsalgorithmen, für andere hingegen nicht.

Übungen zu diesem Kapitel

Übung 12.1 Greedy-Heuristik für Vertex-Cover, schwer

Beweisen Sie, dass die Greedy-Heuristik von Seite 12-26 eine Worst-Case-Approximationsgüte von $\Theta(\ln n)$ aufweist. Gehen Sie dazu wie folgt vor:

1. Finden Sie für jedes n einen Graphen G_n, so dass die Güte der Ausgabe des Greedy-Algorithmus in $\Omega(\ln n)$ liegt.
   Tipp: Benutzen Sie einen bipartiten Graphen als G_n. Ein Ufer von G_n sollte die Größe n haben. Das andere Ufer sollte aus $n-1$ Blöcken $B_2, B_3, \ldots, B_n$ von Knoten bestehen, wobei B_i Größe $\lfloor n/i \rfloor$ hat und jeder Knoten in B_i zu i unterschiedlichen Knoten des ersten Ufers verbunden ist.
2. Zeigen Sie, dass die Greedy-Heuristik immer eine Lösung findet, deren Güte höchstens $\ln n$ ist.
   Tipp: Sei d der maximale Grad in einem Graphen G. Dann musss eine minimale Knotenüberdeckung C von G mindestens $|E|/d \leq |C|$ Knoten enthalten. Ist E_i die Menge an Kanten, die nach Runde i noch vorhanden sind und d_i der maximale Knotengrad nach Runde i, so wählt Greedy immer einen Knoten vom Grad d_i. Damit gilt aber $|E_{i+1}| \leq |E_i| - d_i \leq |E_i| - |E_i|/|C| = |E_i|(1-1/|C|)$. Hieraus folgt $|E_i| \leq |E|(1-1/|C|)^i$. Um den Beweis abzuschließen, nutzen Sie $(1-x)^n \leq e^{-nx}$.

13-1

Kapitel 13
Entwurfsmethode: Fixed-Parameter

Wie man Vertex-Cover für Milliarden von Knoten und Kanten löst

13-2

Lernziele dieses Kapitels

1. Konzept des Fixed-Parameter-Algorithmus verstehen
2. Fixed-Parameter-Algorithmus für das Vertex-Cover-Problem kennen
3. Konzept der Kernelisierung verstehen
4. Kernelisierungs-Algorithmus für das Vertex-Cover-Problem kennen

Inhalte dieses Kapitels

Worum es heute geht

Neulich bei einer Tasse Cappuccino belauschte ich am Nebentisch ein Informatikerehepaar, er Theoretiker, sie Praktikerin.[1] Sie diskutierten die Frage, ob das Problem PLANAR-3-COLORABLE ein »schweres« Problem sei.

Theoretiker Liebling, ich habe neulich zeigen können, dass das planare 3-Färbbarkeitsproblem ein wirklich schwieriges Problem ist.

Praktikerin *(eher auf ihr Himbeertörtchen konzentriert)* Ach ja?

Theoretiker *(sichtlich erfreut über die Nachfrage)* Ja, das Problem ist nämlich NP-vollständig!

Praktikerin *(schiebt sich ein Stück Himbeertörtchen in den Mund)* Hmmpf?

Theoretiker *(interpretiert dies als Aufforderung, auszuholen)* Also, ich habe zeigen können, dass es eine Reduktion von 3-COLORABLE auf dieses Problem gibt. Wie du ja weißt, ist 3-Färbbarkeit ein klassisches NP-vollständiges Problem und folglich genügt es, hiervon eine Reduktion anzugeben, um die Vollständigkeit zu zeigen.

Praktikerin *(trinkt noch einen Schluck Kaffee)* Und damit ist das Problem schwer?

Theoretiker *(etwas verunsichert ob der Frage)* Ja, natürlich, Schatz. Du weißt doch, NP-vollständige Probleme sind schwer, denn wenn wir eines von ihnen in polynomieller Zeit lösen könnten, dann könnten wir ja *alle* Problem in NP in polynomieller Zeit lösen, was ja noch niemand geschafft. Man könnte, sagen wir, sogar das Faktorisierungsproblem in polynomieller Zeit lösen – und dann würde die gesamte IT-Sicherheitsinfrastruktur dieses Planeten in sich zusammenbrechen!

Praktikerin *(schiebt den Rest ihres Himbeertörtchens zur Seite und schaut ihn an)* Na und?

Theoretiker *(völlig entgeistert)* Wie »na und«?

Praktikerin *(trinkt noch einen Schluck Kaffee)* Du meinst, man kann das planare 3-Färbbarkeitsproblem nicht effizient lösen, weil dann das Faktorisierungsproblem als Problem in NP effizient lösbar wäre?

[1] Es sei an dieser Stelle angemerkt, dass meine Rechtschreibkorrektur zwar das Wort »Theoretikerin« kennt, nicht jedoch das Wort »Praktikerin«, was gendersensibilisierte Menschen sicherlich aufhorchen lässt.

Theoretiker *(indigniert)* Das »meine« ich nicht nur, dass kann ich beweisen!

Praktikerin Lass mich dir mal zeigen, was da *wirklich* passiert: Was sind denn die Eingaben beim Faktorisierungsproblem?

Theoretiker Na, Zahlen von vielleicht 2000 Bit Länge, die faktorisiert werden sollen.

Praktikerin Genau. Und wie geht dein Argument nun weiter?

Theoretiker Das Faktorisierungsproblem ist ein typisches Problem in NP: Eine nichtdeterministische Turing-Maschine kann zunächst einen Faktor raten, dann überprüfen, ob dies tatsächlich einer ist, und dann akzeptieren, wenn ein bestimmtes Bit des Faktors 1 ist. Die Laufzeit ist polynomiell und, wenn man sich geschickt anstellt, vielleicht sogar linear. Nach dem Satz von Cook können wir jedes Problem in NP auf CIRCUIT-SAT reduzieren und von dort aus weiter.

Praktikerin In der Tat. Wie groß wird denn der Schaltkreis bei der Reduktion auf CIRCUIT-SAT?

Theoretiker *(grübelt kurz)* Nun, der Schaltkreis hat quadratische Größe in der Laufzeit des Algorithmus.

Praktikerin Also mindestens 4.000.000 bei einer Eingabelänge von 2000 Bits?

Theoretiker Öh, ja. Wohl eher noch etwas mehr, da wir ja noch die Boxen aus der Reduktion brauchen. So eher 100.000.000.

Praktikerin Und dann?

Theoretiker Nun, der Schaltkreis wird in eine 3-SAT Formel umgewandelt grob derselben Größe, also mit etwa 100 Millionen Klauseln.

Praktikerin Und dann?

Theoretiker Dann auf NAE-3-SAT, wobei jede Klausel durch vier Klauseln ersetzt wird.

Praktikerin Und dann?

Theoretiker Dann auf 3-COLORABLE, wobei jede Klausel durch drei Knoten und jede Variable durch zwei Knoten ersetzt wird.

Praktikerin Und dann?

Theoretiker Dann wird noch jede Kreuzung durch ein Gadget ersetzt, wodurch je elf neue Knoten entstehen. Dann ist aber Schluss!

Praktikerin Lass mich das mal zusammenfassen: Wenn ich also wissen möchte, wie die Faktorisierung einer 2000-Bit-Zahl aussieht, kann ich stattdessen versuchen zu entscheiden, ob ein bestimmter planare Graph 3-färbbar ist. Und wie viele Knoten hat der?

Theoretiker *(nuschelt)* Nun, so um die 10 Milliarden, denke ich.

Praktikerin Siehst du, deshalb habe ich vorhin »Na und?« gesagt. Solche gigantischen Eingaben mit einer auch noch extrem konstruierten Form kommen doch in der Praxis gar nicht vor!

Theoretiker Ha! Da kann ich aber auch antworten: »Na und?« Schließlich *könnten* sie vorkommen.

Praktikerin Mag sein. Aber dein schönes NP-Vollständigkeitsresultat widerspricht überhaupt nicht der Möglichkeit, dass es einen Algorithmus gibt, der bei allen *in der Praxis* vorkommenden Eingaben schnell ist. Wie wir gerade gesehen haben, könnte es doch sein, dass wie für planare Graphen mit, sagen wir, maximal einer Million Knoten das 3-Färbbarkeitsproblem in einer Stunde lösen könnte. So viel zum Thema »PLANAR-3-COLORABLE ist ein wirklich schwieriges Problem«…
Bringst du das Geschirr weg?

Die Diskussion stimmte mich nachdenklich. Kann man das planare 3-Färbbarkeitsproblem wirklich für beliebige Graphen mit einer Million Knoten schnell lösen?

Tatsächlich kennt man keinen solchen Algorithmus, jedoch ist viel Wahres an dem Argument der Praktikerin: Wenn ein Problem NP-vollständig ist, heißt das nur, dass es *sehr große und sehr konstruierte Eingaben gibt, die schwer zu lösen sind.* In aller Regel werden das aber gerade *nicht* Eingaben sein, die man »in der freien Wildbahn« antrifft. Die systematische Untersuchung von Eingaben, die »in der Praxis« vorkommen, hat gleich eine ganze Reihe von Theorie-Teilgebieten produziert: Average-Case-Analysen, Smoothed-Analysis und

Fixed-Parameter-Methoden drehen sich alle um die Frage, wie schnell Probleme gelöst werden können, wenn die Eingaben »in der Praxis oft vorkommen«.

In diesem Kapitel wollen wir uns ein besonders erfolgreiches Gebiet, die Fixed-Parameter-Algorithmen, etwas genauer anschauen. Es sei aber gleich vorneweg darauf hingewiesen, dass sich *Theoretiker* hier eine Definition von *»kommt in der Praxis häufig vor«* ausgedacht haben. Dass dabei nicht unbedingt etwas übermäßig Pragmatisches herauskommt, versteht sich fast von selbst. Dass die Theoretiker mit ihrer Definition aber nicht ganz daneben liegen, zeigt der Erfolg dieser Ansätze: Für bestimmte Probleme, insbesondere für das Vertex-Cover-Problem, lassen sich damit tatsächlich Eingaben aus der Praxis mit Hunderten oder gar Tausenden von Knoten lösen.

13.1 NP-Vollständig heißt nicht »unlösbar«

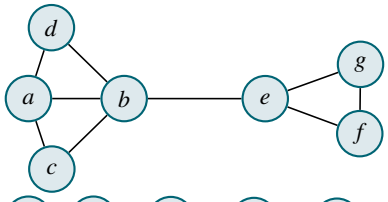
13.1.1 Sind »schwere« Probleme »schwer«?

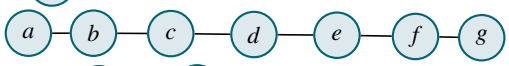
13-4

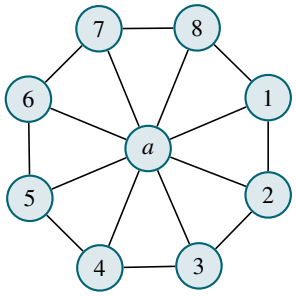
Wie »schwer« sind »schwere« Probleme wirklich?

Sind planare 3-Färbbarkeit und Vertex-Cover schwer?

Welche der folgenden Graphen sind 3-färbbar? Welche haben ein Vertex-Cover der Größe 3?

1.


2.


3.


Ist Subset-Sum schwer?

Können Sie einige der folgenden Zahlen unterstreichen, so dass die Summe der unterstrichenen Zahlen genau 1.000.000.000 beträgt?

- 1
- 12
- 123
- 1234
- 12345
- 123456
- 1234567
- 12345678
- 123456789
- 1234567890

Und nochmal: Ist Vertex-Cover schwer?

Betrachten Sie ein »Riesenrad« (ein Zyklus mit einem Knoten in der Mitte, der mit allen Knoten verbunden ist) mit einer Million Knoten. Besitzt dieser Graph ein

- Vertex-Cover der Größe 3?
- Vertex-Cover der Größe 4?
- Vertex-Cover der Größe 5?
- Vertex-Cover der Größe 500.000?

Wieso lassen sich schwere Problem manchmal sehr schnell lösen?

13-5

Wir haben eben für mehrere NP-*vollständige Probleme* (PLANAR-3-COLORABLE, VC und SUBSETSUM) konkrete Eingaben *sehr schnell per Hand gelöst*.
Mögliche Gründe hierfür könnten sein:

1. Die Eingaben sind *zu klein*, als dass sich die »Schwere« der Probleme einstellt. (Aber: Was ist mit dem Riesenrad?)
2. Die Eingaben sind *zu speziell*, als dass sich die »Schwere« der Probleme einstellt.

Big Ideas

Strategien zur Lösung schwieriger Probleme:

- Betrachte nur »durchschnittliche« Eingaben. Die führt zur *Average-Case-Analyse*.
- Betrachte nur Eingaben, die »leicht verrauscht« sind. Die führt zur *Smoothed-Analysis*.
- Betrachte nur Eingaben, die »in der Praxis« vorkommen. Die führt zu *Fixed-Parameter-Algorithmen*.

13.1.2 Fallbeispiel: Vertex-Cover

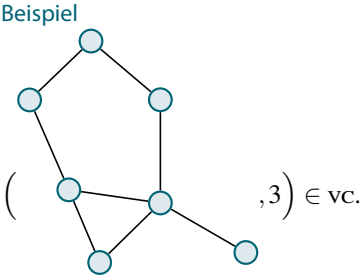
Ein schweres Problem, das wir schnell lösen wollen.

13-6

▶ Definition: Vertex-Cover

Die Sprache VC enthält alle Paare (G,k), wobei G ein ungerichteter Graph ist mit n Knoten und m Kanten, k eine Zahl ist und es in G eine Teilmenge U der Knoten gibt mit $|U| \leq k$, so dass jede Kanten von G wenigstens einen ihrer Endpunkte in U hat.

Beispiel



$(\quad ,3) \in$ VC.

Dieses Problem ist aus verschiedenen Gründen interessant:

- Falls die Kanten des Graphen »Konflikte« repräsentieren, so ist ein Vertex-Cover eine (kleine) Knotenmenge, die man entfernen kann, um alle Konflikte aufzulösen.
- Falls die Kanten des Graphen »Effekte« repräsentieren, so ist ein Vertex-Cover eine (kleine) Knotenmenge, die an allen Effekten beteiligt ist.
- Das Komplement eines minimalen Vertex-Cover ist eine maximale unabhängige Menge.

13.1.3 Ein ehrgeiziges Ziel

Ein *sehr* ehrgeiziges Ziel.

13-7

Unser prinzipielles Ziel

Ein *schneller, exakter* Algorithmus, der zwar *exponentielle Zeit* braucht, aber nur auf *sehr konstruierten Eingaben* und auf *normalen Eingaben* sehr schnell ist.

Ein konkretes Ziel

Es ist ein Algorithmus gesucht, der für *alle* Graphen mit $n = 1000$ Knoten, $m = 10000$ Kanten und $k = 80$ auf einem normalen Computer in wenigen Minuten ein Vertex-Cover findet.

13.2 Der Fixed-Parameter-Ansatz

13-8

Ein erster Versuch, das Vertex-Cover-Problem zu lösen

Brute-Force-Algorithmus

```
1  input G = (V,E), k
2  foreach C ⊆ V with |C| = k do
3    if C is a vertex cover of G then
4      output ''size-k vertex cover exists'' and stop
5  output ''no size-k vertex cover exists''
```

Die Laufzeit ist $O(n^k)$, was für Werte wie $n = 1000$ and $k = 80$ *phantastisch lange dauert*. Bis zum Jahr 1988 glaubt man, dass dies die beste Methode sei, VC zu lösen..

13-9

Ein zweiter Versuch, basierend auf Backtracking.

Backtracking-Algorithmus

```
1  procedure vc-backtrack (G, k)
2      if k < 0 then
3          return false
4      else if G has no edges then
5          return true
6      else
7          foreach v ∈ V do
8              G' ← G without the vertex v
9              if vc-backtrack (G', k − 1) then
10                 return true
11         return false
```

Falls es ein Vertex-Cover gibt, so hängt die Laufzeit nun von den Details der Eingabe ab. Gehen wir die Knoten $v \in V$ in einer »geschickten« Reihenfolge durch (beispielsweise beginnend mit den Knoten maximalen Grades), so *können wir eventuell* schnell ein Vertex-Cover finden. Jedoch gilt: Falls es kein Vertex-Cover der Größe k gibt, so ist die Laufzeit immernoch $O(n^k)$.

13.2.1 Der Pakt mit dem Teufel

13-10

Downey und Fellows gehen einen Pakt mit dem Teufel ein.

Autor Julius Nisle

Was der Teufel will. . .

Der Teufel erlaubt uns nur, *Exponentialzeit-Algorithmen* für VC zu entwerfen. (Oder P = NP zu beweisen.)

. . . das soll er bekommen

Wir entwerfen einen Algorithmus, der *zwar exponentielle Zeit braucht*, aber *in Bezug auf k und nicht in Bezug auf n oder m.*

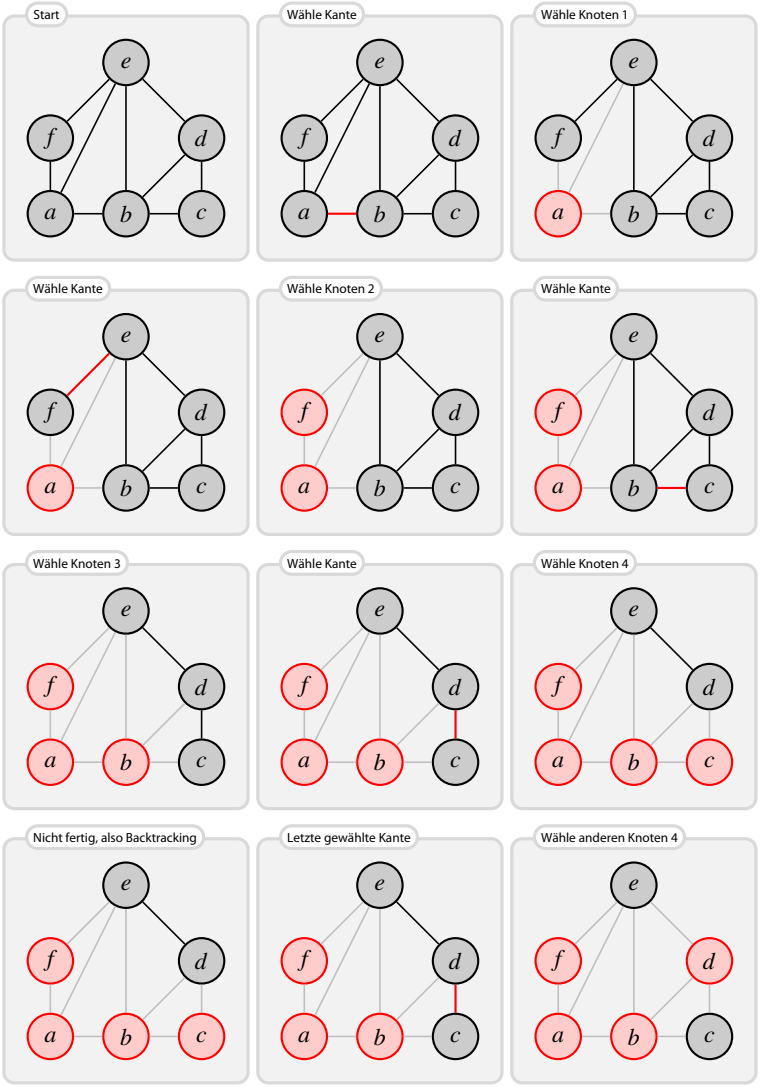
13.2.2 Eine brillante Idee

Eine einfache, aber brillante Idee: Backtracking entlang der *Kanten*! 13-11

Fellows Algorithmus

```
1  procedure vc-backtrack-edges (G, k)
2      if k < 0 then
3          return false
4      else if G has no edges then
5          return true
6      else
7          pick any edge {x,y} ∈ E
8          foreach v ∈ {x,y} do
9              G' ← G without the vertex v
10             if vc-backtrack-edges (G', k − 1) then
11                 return true
12         return false
```

Der Algorithmus von Fellows für $k = 4$. 13-12



13-13 Die Laufzeit von Fellows Algorithmus.

▶ Satz
Der Algorithmus ist korrekt und arbeitet in Zeit $O(2^k m)$.

Beweis. Die Korrektheit der Ausgabe sieht man durch Induktion über die Anzahl m an Kanten in G:

- Hat G keine Kanten, so ist die Ausgabe korrekt.
- Hat G eine Kante $\{x,y\} \in E$, dann hat G ein Vertex-Cover der Größe k genau dann, wenn G ohne x oder G ohne y ein Vertex-Cover der Größe $k-1$ hat.

Betreffend die Laufzeit argumentieren wir wie folgt: Die Laufzeit genügt offenbar der Formel $T(k) = 2T(k-1) + O(m)$ und $T(0) = O(m)$. Man überzeigt sich leicht, dass dies die Lösung $T(k) = \Theta(2^k m)$ hat. □

13.2.3 Verfeinerungen der Idee

13-14 Zwei einfache, aber folgenreiche Beobachtungen.

Beobachtung 1: Graphen ohne Verzweigungen sind einfach
Nehmen wir an, ein Graph hat *keine Knoten von Grad 3 oder höher*. Dann besteht er ausschließlich aus *Kreisen und Pfaden*. Für solche Graphen ist es aber *leicht*, ein optimales Vertex-Cover zu berechnen:

- Man benötigt für ein Pfad der Länge n genau $\lfloor n/2 \rfloor$ Knoten, um alle seine Kanten zu überdecken.
- Man benötigt für einen Kreis der Länge n genau $\lceil n/2 \rceil$ Knoten, um alle seine Kanten zu überdecken.

Beobachtung 2: Ein Knoten oder seine Nachbarn
Für jeden Knoten v eines Graphen und jedes Vertex-Cover C des Graphen gilt:

- v ist ein Element von C oder
- alle Nachbarn von v sind Elemente von C oder
- beides ist der Fall.

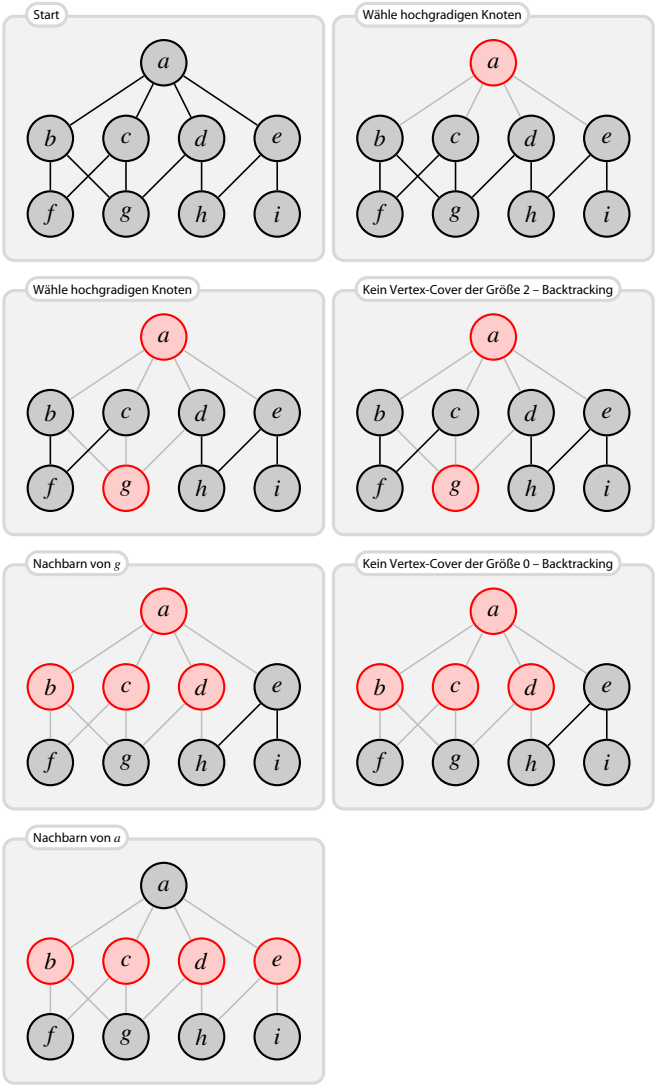
13-15 Backtracking entlang *»hochgradiger Knoten«*.

Hochgradiges Backtracking

```
1  procedure vc-backtrack-high-degree (G, k)
2      if k < 0 then
3          return false
4      else if G has no vertex of degree at least 3 then
5          compute the size s of the smallest vertex cover of G
6          return k ≥ s
7      else
8          pick a node x ∈ V of degree 3 or more
9          N ← {v | {x,v} ∈ E}, that is, the neighbors of x
10         foreach S ∈ {{x},N} do
11             if vc-backtrack-high-degree (G − S, k − |S|) then
12                 return true
13         return false
```

Das hochgradige Backtracking für $k = 4$.

13-16



Die Geschwindigkeit des hochgradigen Backtrackings.

13-17

▶ Satz

Der Algorithmus ist korrekt und arbeitet in Zeit $O(1.49535^k nm)$.

Beweis. Die Korrektheit sollte klar sein, betrachten wir also Laufzeit. Die Anzahl $N(k)$ an *rekursiven Aufrufen* ist

$$\begin{aligned} N(k) &\leq N(k-1) + N(k-3) + 1, \\ N(2) &= 1, \\ N(1) &= 0, \\ N(0) &= 0, \end{aligned}$$

da wir zwei rekursive Aufrufe durchführen, einen für $k-1$ und einen für $k - |S| \leq k-3$. Wir behaupten, dass $N(k) \leq 5^{k/4} - 1$ eine Lösung hiervon ist. Dies sieht man so: Für $k \in \{0,1,2\}$ ist dies der Fall. Für den induktiven Schritt von $k-1$ auf k betrachte

$$\begin{aligned} N(k) &\leq N(k-1) + N(k-3) + 1 \\ &\leq 5^{k/4-1/4} + 5^{k/4-3/4} - 1 \\ &= \underbrace{(5^{-1/4} + 5^{-3/4})}_{\approx 0.968 < 1} 5^{k/4} - 1. \end{aligned}$$

Damit ist die Behauptung gezeigt. □

13-18 Das aktuelle State-of-the-Art-Resultat bezüglich Vertex-Cover.

▶ Satz: Chen, Kanj, Xia 2006
Man kann VC *in Zeit* $O(1.2738^k + nm)$ *lösen.*

Man beachte: Es gilt $1.2738^k < 10^9$ für $k \leq 85$ und der Term $O(nm)$ ist *additiv*, nicht *multiplikativ*. Dies bedeutet, dass VC für jeden Graphen mit $n = 1000$, $m = 10000$ und $k = 80$ auf einem normalen Computer *in Sekundenbruchteilen (!) gelöst werden kann*.

13.2.4 Das allgemeine Konzept

13-19 Das allgemeine Konzept des Fixed-Parameter-Ansatzes

- Der *Pakt mit dem Teufel* hat für Vertex-Cover einen Algorithmus ergeben mit folgender Laufzeit:

$$\underbrace{c_1^k}_{\text{exponentielle Abhängigkeit von } k} \cdot \underbrace{(nm)^{c_2}}_{\text{polynomielle Abhängigkeit von } n \text{ und } m}.$$

- Ist der Parameter k *fest*, so erhält man einen Polynomialzeitalgorithmus in Bezug auf alle anderen Parameter.

Fixed-Parameter-Algorithmen allgemein
Fixed-Parameter-Algorithmen haben immer eine Laufzeit von

$$f(k) \cdot l^c,$$

wobei

1. f eine beliebige Funktion ist (typischerweise so etwas wie 2^k),
2. k ein problemspezifischer Parameter, der bei typischen Eingaben *konstant* ist,
3. l die Eingabelänge ist und
4. c eine Konstante ist.

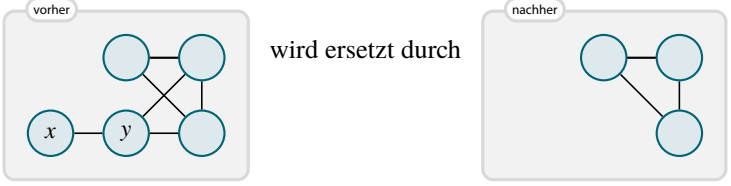
13.3 Kernelisierung

13-20 Die Idee der Vorverarbeitung.
»Schwere« Probleme sind nur bei *großen Eingaben* »schwer«. Folglich könnte man versuchen, große Eingaben *immer wieder durch kleinere zu ersetzen,* die genau dann Elemente der Sprache sind, wenn die großen Eingaben es waren. Diesen Vorgang nennt man »Kernelisierung«. Wenn eine Eingabe nicht weiter verkleinert werden kann, nennt man sie einen *Kern*.

Beispiel: Eine mögliche Ersetzungsregel für Vertex-Cover

- In einer Eingabe (G,k) sei $x \in V$ ein Knoten vom Grad 1.
- Dann kann (G,k) ersetzt werden durch $(G',k-1)$, wobei in G' sowohl x wie auch sein (einziger) Nachbar entfernt wurde.

Diese Regel ist korrekt, denn statt x kann man auch immer den Nachbarn von x in ein Vertex-Cover aufnehmen.



Ein Kernelisierungsalgorithmus für Vertex-Cover.

13-21

▶ Satz

Auf eine Eingabe (G,k) wende man folgende Regeln an, bis es nicht mehr geht:

1. *Falls G einen isolierten Knoten hat, entferne ihn.*
2. *Falls G einen Knoten v vom Grad mindestens $k+1$ hat, ersetze die Eingabe durch $(G-v,k-1)$.*

Sei (G',k') das Resultat. Dann gilt:

1. *$(G,k) \in$ vc genau dann, wenn $(G',k') \in$ vc.*
2. *Falls G' noch mehr als $k(k+1)$ Knoten hat, so gilt $(G',k') \notin$ vc.*

Beweis. Die erste Behauptung ist leicht einzusehen. Für die zweite nehmen wir an, dass G' tatsächlich mehr als $k(k+1)$ Knoten hat: Da die Regeln nicht mehr anwendbar sind, gibt es keine Knoten vom Grad $k'+1$ in G'. Wählt man also beliebige $k' \leq k$ Knoten aus G, so ergeben diese zusammen mit ihren Nachbarn maximal $k(k+1)$ Knoten. Da es in G' aber mehr Knoten geben soll und diese nicht isoliert sind (denn die erste Regel ist nicht anwendbar), folgt, dass es nicht abgedeckte Kanten in G' gibt. □

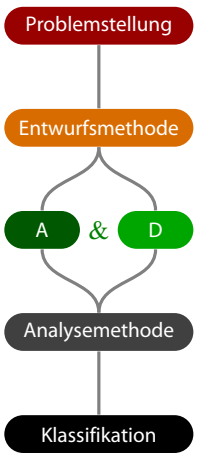
▶ Folgerung

Man kann vc *in Zeit $O(|G|+1{,}49535^k k^2)$ lösen.*

Mit anderen Worten: *Das* NP*-vollständige Problem »Vertex-Cover« kann effizient gelöst werden für beliebig große Eingaben, so lange $k \leq 85$ gilt.*

Zusammenfassung dieses Kapitels

13-22



Vertex-Cover	Vertex-Cover
Fixed-Parameter-Ansatz	Kernelisierung und Fixed-Parameter-Ansatz
Hochgradiges Backtracking	Kernelisierung und hochgradiges Backtracking
Rekursionsgleichung	Rekursionsgleichung
Laufzeit $O(1{,}49535^k mn)$	Laufzeit $O(\|G\|+1{,}49535^k k^2)$

▶ Fixed-Parameter-Ansatz

Ein *Fixed-Parameter-Algorithmus* für ein Problem löst dieses in Zeit

$$f(k) \cdot l^c,$$

wobei f eine beliebige Funktion ist, k ein problemspezifischer Parameter, l die Eingabelänge und c eine Konstante.

▶ Beispiele

Beispiele, für die man Fixed-Parameter-Algorithmen kennt:

- Vertex-Cover
- Tripartites Matching
- Planares Dominating-Set

Jedoch gibt es auch Probleme, für die man vermutet, dass es *keine* Fixed-Parameter-Algorithmen für sie gibt:

- Clique
- Dominating-Set
- Färbbarkeit

▶ Kernelisierung

Unter der *Kernelisierung* einer Eingabe x für ein Problem P versteht man die wiederholte Ersetzung von x durch kürzere Eingaben x', so dass:

1. $x \in P$ genau dann, wenn $x' \in P$ und
2. die Größe des letzten x' ist durch einen problemspezifischen Parameter beschränkt.

Motto: Kernelisierung schadet nie.

Zum Weiterlesen

[1] Jörg Flum und Martin Grohe, *Parametrized Comlexity Theory,* Springer, 2006.

Ein aktuelles Buch zum Thema für alle, die es etwas genauer wissen wollen.