RANDOMISIERTE ALGORITHMEN I.

ENTWURFSMETHODEN

„ENTWURFSMETHODEN" bedeutet hier eher Paradigmen, Prinzipien, Sichtweisen; bzw. die verschiedenen Gründe warum Randomisierung verwendet wird (z.B. „Vermeidung von Worst-Case" oder „Randomisierung in Konfliktsituationen").

Es gibt keine scharfe Grenzen zwischen diesen Sichtweisen: oft treffen mehr als eine Sichtweisen zu.

A. VERMEIDUNG VON WORST-CASE EINGABEN

(Das zufällige (theoretische) Permutieren der Eingabe am Anfang vom QUICKSORT oder vom Algorithmus für das Sekretär-Problem, dient der Vermeidung von Worst-Case Eingaben, (die für jeden konkreten deterministischen Algorithmus einfach(er) konstruierbar sind).)

Die Auswertung von Spielbäumen
(game-tree evaluation)

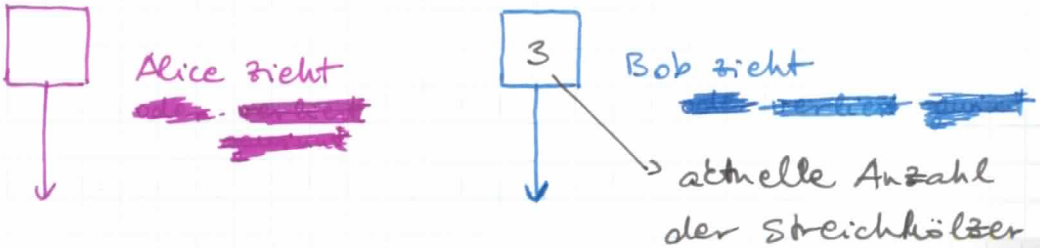
Beispiel: Alice und Bob spielen das folgende Spiel:
Sie legen abwechselnd 1 oder 2 Zündhölzer auf den Tisch. Der Spieler der das 5-te Zündholz gelegt hat, verliert. Alice fängt an.

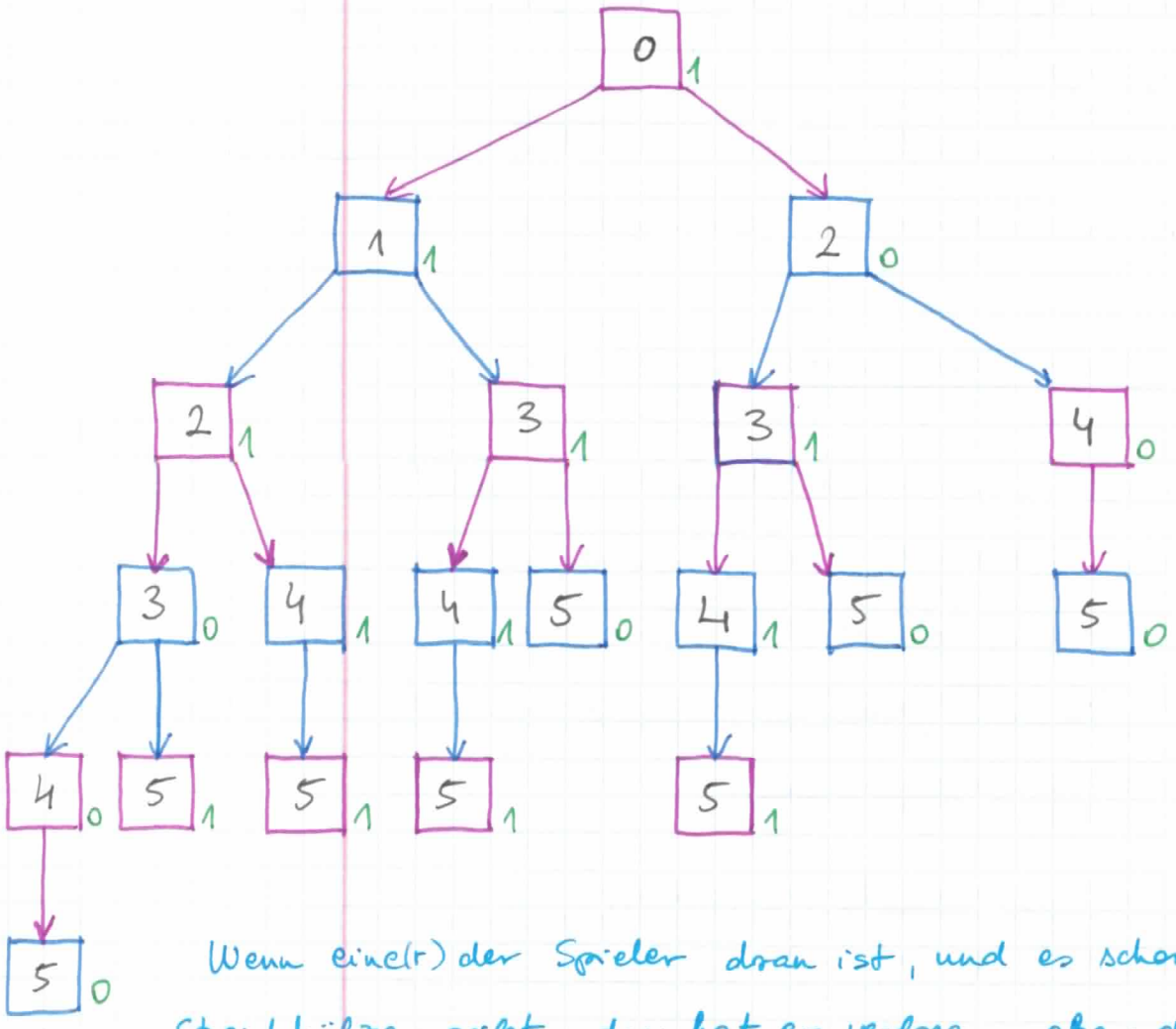
Wer von den beiden hat hier eine Gewinnstrategie?

R2.

Wir werden die möglichen Zügen der Spieler (und dann den Gewinn von Alice bottom-up) mit einem Spielbaum darstellen:

Notation:





Wenn eine(r) der Spieler dran ist, und es schon 4 Streichhölzer gibt, dann hat er verloren; aber dann, andererseits falls schon 2 oder 3 auf dem Tisch liegen, hat er/sie „gewonnen"; also sollte sie/er so anfangen, dass der Gegner auf 2 oder 3 ergänzen muss → sie/er soll mit 1 anfangen!

(Wer hat eine Gewinnstrategie, falls das 4-te Streichholz verliert?)

Wir beschriften die Blätter und dann rekursiv bottom-up alle Knoten des Baums mit 1 oder 0:
1 bedeutet, dass Alice, der erste Spieler gewonnen hat, bzw. eine Gewinnstrategie hat; 0 bedeutet, dass Bob der zweite Spieler gewonnen hat, bzw. eine Gewinnstrategie hat.
Die Beschriftung der Wurzel verrät dann, wer am Anfang eine Gewinnstrategie hat (das Spiel verläuft dann entsprechend nur über 1-Knoten oder nur über 0-Knoten).

Alice möchte also den Wert der Beschriftung im aktuellen Knoten (also 1 oder 0) während des Spiels maximieren (1-Knoten wählen), und Bob möchte ihn minimieren.

Wir nennen sie entsprechend MAX-Spieler bzw. MIN-Spieler.

Wie werden die Beschriftungen (Spielwerte) bottom-up berechnet?

Allgemeine Beschreibung des Problems:

→ jedes (mehrzügige) zwei-personen Spiel mit beschränkt langen Spielen besitzt einen Spielbaum T

→ Knoten mit geradem Abstand von der Wurzel sind MAX-Knoten in denen der MAX-Spieler (Alice) am Zug ist.

→ Knoten mit ungeradem Abstand von der Wurzel sind MIN-Knoten in denen der MIN-Spieler (Bob) am Zug ist.

R4.

Eingabe: Ein Spielbaum T mit einem Spielwert in jedem Blatt, d.h. der Auszahlung an den MAX-Spieler (Alice) wenn das Spiel in diesem Blatt endet. (Blätter haben ja keine ausführende Kanten/Kinderknoten im Spielbaum)

Aufgabe: Die Auswertung des Spielbaums T: den Spielwert der Wurzel zu bestimmen, also die Auszahlung an Alice (die sie in jedem Fall erreichen kann, egal wie Bob spielt).

Die Auswertung von Spielbäumen spielt eine wichtige Rolle für Spiel-Programme. Da für die Auswertung normalerweise exponentielle Laufzeit benötigt wird, haben wir (zum Glück!) sehr schwierige Probleme. Denk an Schach-Programme...

Definition: Der Spielwert eines beliebigen MAX-Knotens ist das Maximum der Spielwerte seiner Kinder-Knoten.

(die Auszahlung die Alice mit optimalen Spielzügen in jedem Fall erreichen kann)

Der Spielwert eines MIN-Knotens ist das Minimum der Spielwerte seiner Kinder-Knoten.

(die minimale Auszahlung an Alice die Bob mit seinen optimalen Spielzügen aus dieser Stellung in jedem Fall erreichen kann)

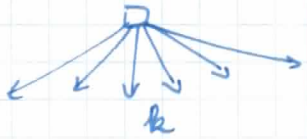
Untere Schranke für deterministische Algorithmen

Ein deterministischer Algorithmus kann alle Knoten des Baumes T bottom-up auswerten bis zur Wurzel. Hierfür muss er aber im schlimmsten Fall jedes Blatt (dessen Spielwert) abfragen. Diese Behauptung zeigen wir zuerst:

Vereinfacht nehmen wir an, dass

→ der Spielwert in jedem Knoten 0 oder 1 ist

→ T ein k-ärer Baum ist, d.h. jeder innere Knoten genau k Kinder hat.



→ T ein vollständiger Baum der Tiefe 2t ist d.h. jeder Spieler t Züge hat.

→ wir bezeichnen einen solchen Spielbaum mit T_k^t

Theorem: Sei A ein deterministischer Algorithmus, der den Baum T_k^t durch Abfragen der Blattwerte auswertet. Dann gibt es eine Eingabe (Wertezuweisung der Blätter), so dass A alle Blätter (also k^{2t} Blätter!) inspizieren muss.

Beweis: Wie sieht also eine Eingabe aus? Eine Eingabe ist eine Zuweisung von 0 oder 1 zu jedem der k^{2t} Blätter. Das ist exponentiell groß in der Anzahl der Spielzüge. Wir zeigen jetzt, dass ein deterministischer Algorithmus im schlimmsten Fall all diese k^{2t} viele Werte betrachten soll um die Wurzel korrekt auszuwerten ...

RG.

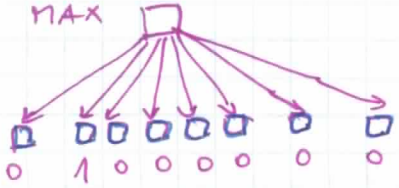
→ der Alg A fragt die Spielwerte der Blätter ab in irgendeiner deterministischen Reihenfolge (die von früher abgefragten Werten abhängt), bis er mit Sicherheit den Wert der Wurzel, 0 oder 1, bestimmen kann.

→ Idee: Ein Gegner beantwortet jede Abfrage mit 0 oder 1 so dass A alle Blätter abfragen muss um entscheiden zu können.

Der Beweis geht mit Induktion über die Tiefe des Baumes (genauer: über t)

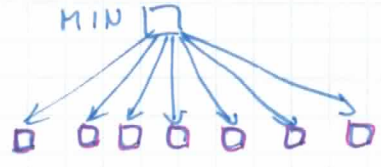
Beispiel (~~Basisschritt~~):

Sei die Tiefe 1 und die Wurzel ein MAX-Knoten



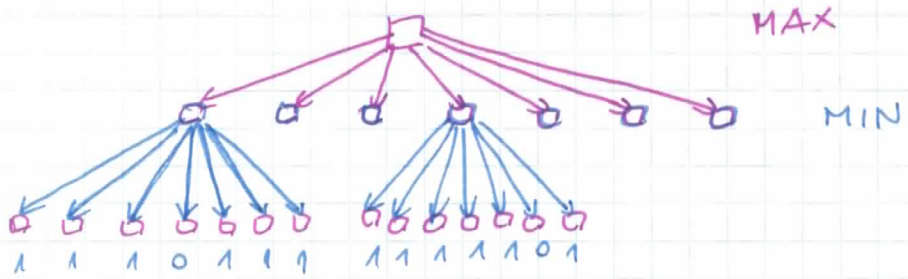
→ der Gegner antwortet mit 0 bis zum zuletzt-abgefragten Kind; am Ende beliebig mit 0 oder 1

Sei die Tiefe 1 und die Wurzel ein MIN-Knoten



→ der Gegner beantwortet die Abfragen mit 1 bis zum zuletzt-gefragten Kind.

Basisschritt $t = 1$ Tiefe $= 2t = 2$



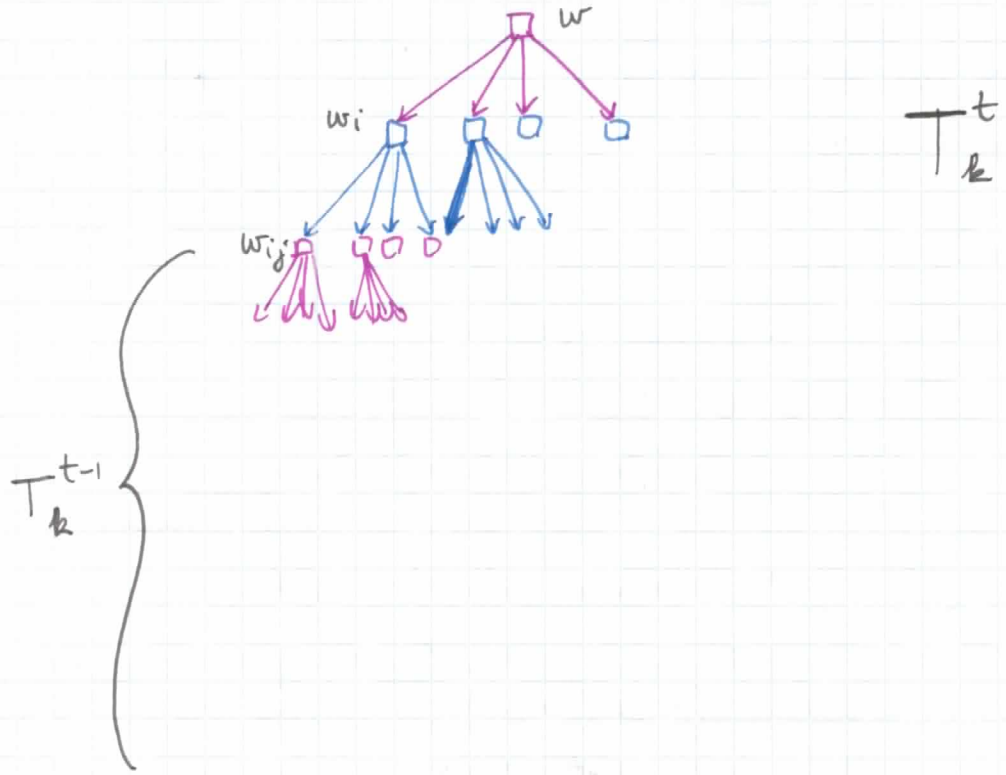
T_k^1

→ der Gegner antwortet mit 1 bis zum letzten Kind eines jeden MIN-Knotens; das letzte Kind beantwortet er mit 0.

→ so müssen alle Kinder von jedem MIN-Knoten, also alle Blätter abgefragt werden

Induktionsschritt $(t-1) \longrightarrow t$

Induktionsannahme: Für einen Spielbaum T_k^{t-1} der Tiefe $2(t-1)$ müssen alle Blätter abgefragt werden um den Wert in der Wurzel entscheiden zu können.



Jedes Enkelkind w_{ij} der Wurzel w ist die Wurzel eines $T_k^{t-1} = T_{ij}$. Der Gegner beantwortet die Abfragen der Blätter so, dass der Wert des w_{ij} bis zum letzten Blatt von T_{ij} offen bleibt, und dann so dass der Wert von w_{ij} 1 wird; ausser wenn w_{ij} das letzte Kind von w_i ist, dann soll der Wert von w_{ij} 0 werden. □

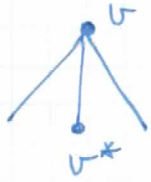
Fazit: Jeder deterministische Algorithmus hat $\Omega(k^{2t})$ Laufzeit im Worst-Case.

(Es gibt für den Algorithmus mindestens eine Eingabe — die wir für den Gegner eben definiert haben — für die der Algorithmus k^{2t} Abfragen braucht.)

Der folgende Algorithmus hat eine viel bessere erwartete Laufzeit für jede fixierte Eingabe:

Ein randomisierter Algorithmus

Die Funktion Wert(v) wird rekursiv berechnet:



Wert(v):

FALLS v ein MAX-Knoten

REPEAT

→ wähle zufällig ein neues Kind von v
sei v^* dieses Kind

→ berechne Wert(v^*) rekursiv

→ FALLS Wert(v^*)=1, gib Wert(v)=1 aus.

UNTIL (Wert(v^*)=1), oder alle Kinder v^* abgefragt

FALLS alle Wert(v^*)=0, gib Wert(v)=0 aus.

FALLS v ein MIN-Knoten

REPEAT

→ wähle zufällig ein neues Kind v^* von v

→ berechne Wert(v^*) rekursiv

→ FALLS Wert(v^*)=0, gib Wert(v)=0 aus

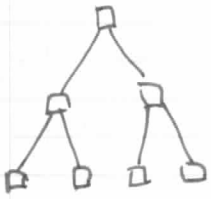
UNTIL (Wert(v^*)=0 oder) alle Kinder abgefragt

FALLS alle Wert(v^*)=1, gib Wert(v)=1 aus

Berechne Wert(w) für w = Wurzel

Wir analysieren die erwartete Laufzeit für binäre Bäume ($k=2$)

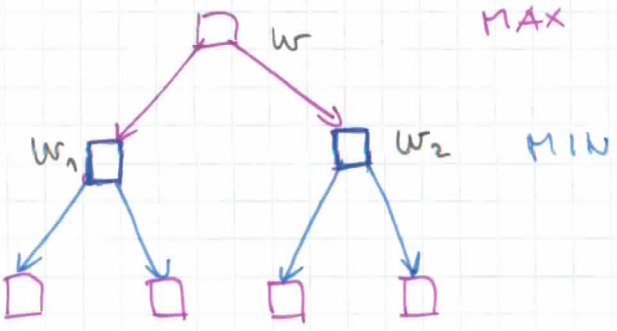
Wir haben gesehen: jeder deterministische Algorithmus inspiziert $\Omega(2^{2t}) = \Omega(4^t)$ Blätter im Worst-Case.



Theorem: Der ~~ran~~domisierte Algorithmus inspiziert für jede Eingabe $\leq 3^t$ Blätter in Erwartung.

Beweis: mit Induktion über die Tiefe des Baumes: (genauer: über t)

Basisschritt $t=1$ Tiefe $= 2t = 2$



Zu zeigen: die erwartete Laufzeit ist ≤ 3

R10

Die Erwartung ist über die zufälligen Wahlen der Kinder im Laufe des Algorithmus, und nicht über die möglichen Spielwerte!

Die folgende Analyse soll für jede Werte-Zuweisung der Blätter, also für jeden Wert(w) separat gelten!

Fall 1: $Wert(w) = 0$

dann ist $Wert(w_1) = 0$ und $Wert(w_2) = 0$

→ w_1 hat mindestens ein Kind mit Wert 0, und mit Prob $\geq \frac{1}{2}$ wird so ein Kind (Blatt) als erstes in $Wert(w_1)$ abgefragt;

→ die erwartete Anzahl von Abfragen in T_{w_1} ist

$$\leq \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 2 = \frac{3}{2}$$

ebenfalls, die erwartete Anzahl von Abfragen in T_{w_2} ist

$$\leq \frac{3}{2}$$

⇒ die erwartete Anzahl von Abfragen in T_w ist die Summe von denen in T_{w_1} und T_{w_2}, also 3.

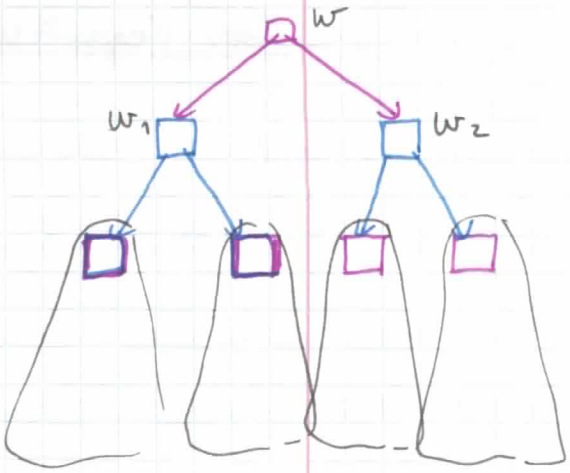
Fall 2: $Wert(w) = 1$

dann ist $Wert(w_1) = 1$ oder $Wert(w_2) = 1$ (oder beide) und mit Prob $\geq \frac{1}{2}$ wird so ein Kind von w als erstes ausgewertet, und kein weiteres Kind mehr.

die erwartete Anzahl abgefragter Blätter ist höchstens

$$\leq \frac{1}{2} \cdot 2 + \frac{1}{2} \cdot 4 = 3$$

Induktionsschritt:



In diesem Fall ist jedes Enkelkind von w die Wurzel eines Baumes T_2^{t-1}. Laut Induktionsannahme braucht die Auswertung einer solchen Wurzel in Erwartung höchstens 3^{t-1} Abfragen. Die Analyse geht jetzt mit analogem Argument wie im Basisschritt:

Fall 1. $Wert(w) = 0$ dann ist die erwartete Anzahl von Abfragen in T_{w_1}

$$\leq \frac{1}{2} \cdot 3^{t-1} + \frac{1}{2} \cdot 2 \cdot 3^{t-1} = \frac{3}{2} 3^{t-1}$$

in T_{w_2}

$$\leq \frac{1}{2} \cdot 3^{t-1} + \frac{1}{2} \cdot 2 \cdot 3^{t-1} = \frac{3}{2} 3^{t-1}$$

insgesamt: $\leq 3^t$

Fall 2. $Wert(w) = 1$ mit Prob $\geq \frac{1}{2}$ wird nur ein Teilbaum T_{w_1} oder T_{w_2} abgefragt, und die erwartete Anzahl der Abfragen ist

$$\leq \frac{1}{2} \cdot 2 \cdot 3^{t-1} + \frac{1}{2} \cdot 4 \cdot 3^{t-1} = 3^t.$$

□

Zur Erinnerung: Zu jeder (zufälligen) Bitfolge des Algorithmus gehört eine Berechnung B, also ein deterministischer Algorithmus. (Jede Bitfolge sei genauso lang wie die zugehörige Berechnung fordert $P_B = \frac{1}{2^k}$ für k lange Folge.) Wir haben somit aus einer Menge von deterministischen Algorithmen B zufällig einen ausgewählt.

Jede Berechnung B hat worst-case Eingaben, aber jede eine andere: keine Eingabe ist worst-case für alle Berechnungen B. Da B zufällig gewählt wird, ist es unwahrscheinlich, dass die aktuelle Eingabe worst-case (oder schlecht) für B sein wird.

Jede Eingabe ist gut „für die meisten" Algorithmen B aus denen zufällig gewählt wird.

R12.

Las Vegas und Monte Carlo

Der randomisierte Algorithmus für die Auswertung von Spielbäumen gibt immer das richtige Ergebnis aus.

Nur die Laufzeit ändert sich in Abhängigkeit von den zufälligen Schritten (von den erhaltenen Zufallsbits).
Solche randomisierte Algorithmen, die sich nie irren, nennt man Las Vegas Algorithmen.

Monte Carlo Algorithmen

Randomisierte Algorithmen haben verschiedene Berechnungen B (für jede Folge von Zufallsbits eine B). Es ist vorstellbar, dass ein Algorithmus, der sich mit mancher seiner Berechnungen irrt, immer noch ein guter, nützlicher Algorithmus sein kann! Wie und wann?

Wir nehmen an, dass wir Entscheidungsprobleme, bzw. Algorithmen haben, die jede Eingabe w entweder akzeptieren oder verwerfen. (für eine Sprache $L \subseteq \Sigma^*$ $w \in L$ oder $w \notin L$ ausgeben)

Akzeptanzwahrscheinlichkeit:

Definition: Ein randomisierter Algorithmus akzeptiert w mit Wahrscheinlichkeit p falls

$$\sum_{\{B: B \text{ akzeptiert } w\}} p_B = p$$

(die Wahrscheinlichkeit akzeptierender Berechnungen)

Beispiel: Ein randomisierter Algorithmus (Alg) soll von einer Eingabezahl N entscheiden ob N eine Primzahl ist.

Falls N Prim: der Algorithmus sagt PRIM mit Wahrscheinlichkeit 1

Falls N kein Prim: der Algorithmus sagt PRIM mit Wahrscheinlichkeit $\leq \frac{1}{2}$

Wie könnte man vorgehen um in dem Ergebnis sicherer zu werden?

(Es gilt wie immer: die Wahrscheinlichkeit $\leq \frac{1}{2}$ bezieht sich auf die gezogenen Zufallsbits, und gilt für jede fixierte Eingabe: wenn wir den Algorithmus für dieselbe Eingabe nochmal laufen lassen, ergibt er unabhängig wieder mit der obigen Wahrscheinlichkeit ein korrektes oder falsches Ergebnis!)

Wir definieren einen neuen Algorithmus Alg_k:

→ Wir führen k Berechnungen mit der Eingabe N unabhängig voneinander aus. Falls der Alg. mindestens einmal „kein Prim" ausgibt, geben wir „kein Prim" aus; [redacted], falls der Alg. k-mal „Prim" ausgibt, geben wir „Prim" aus. Dieser Alg_k hat Fehlerwahrscheinlichkeit $\leq \left(\frac{1}{2}\right)^k$ und irrt sich immer nur in eine Richtung. (das ist die Wahrscheinlichkeit, dass der Alg. für eine Zusammengesetzte Zahl k-mal hintereinander „Prim" ausgibt.) Wir nennen den Algorithmus der aus k unabhängigen Berechnungen von Alg. besteht, Alg_k.

Was ändert sich, wenn die Fehlerwahrscheinlichkeit nicht $\leq \frac{1}{2}$ sondern $\leq \frac{8}{9}$ ist? → Im Wesentlichen nichts!

Die Fehlerwahrscheinlichkeit von Alg_k ändert sich auf $\left(\frac{8}{9}\right)^k$

wobei $\left(\frac{8}{9}\right)^k \to 0$ (wenn $k \to \infty$) gilt genauso wie $\left(\frac{1}{2}\right)^k \to 0$

(es gilt sogar: $\left(\frac{8}{9}\right)^k$ geht gegen 0 mit exponentieller Geschwindigkeit in k, aber die Laufzeit von Alg$_k$ wächst nur linear in k)

Def: A ist ein Monte Carlo Algorithmus mit einseitig beschränktem Fehler für die Sprache L, falls für alle Eingaben w gilt

$$w \in L \implies \text{Prob.}(A \text{ verwirft } w) < \frac{1}{2}$$

$$w \notin L \implies \text{Prob}(A \text{ akzeptiert } w) = 0$$

Die Zahl $\frac{1}{2}$ in dieser Definition ist unwichtig willkürlich. Aus $\frac{8}{9}$ oder irgendeiner Zahl $q < 1$, oder sogar aus Fehlerwahrscheinlichkeit $1 - \frac{1}{\text{poly}(n)}$ können wir $< \frac{1}{2}$ machen indem

k-viele unabhängige Berechnungen ausgeführt werden, für k polynomiell in der Eingabelänge n. $\left(\frac{8}{9}\right)^k < \frac{1}{2}$ bzw. $\left(1 - \frac{1}{\text{poly}(n)}\right)^k < \frac{1}{2}$ ist nötig

Bemerkung: Im obigen Beispiel wäre L die Sprache der zusammengesetzten Zahlen.

Im Folgenden sehen wir ein Beispiel für $1 - \frac{1}{\text{poly}(n)}$

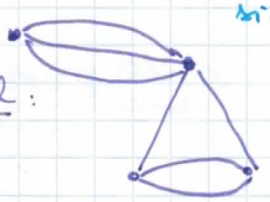
(= mit Fehlerwahrscheinlichkeit $1 - \frac{1}{q(n)}$ für ein Polynom $q(n)$)

Beispiel: Der Min-Cut Algorithmus

(Siehe Motwani-Raghavan 1.1)

Definition: Ein Multigraph $G(V,E)$ kann beliebig viele Kanten zwischen je zwei Knoten $v_1, v_2 \in V$ haben. (In Notation kann man dies z.B. mit der Multiplizität $c(\bar{e})$ der Multi-Kante $\bar{e} \in E$ ausdrücken, wobei $c(\bar{e}) \in \{0,1,2,3,\dots\}$ für jede Kante. (Wir arbeiten lieber mit den einzelnen Kanten sichtbar in den Abbildungen:))

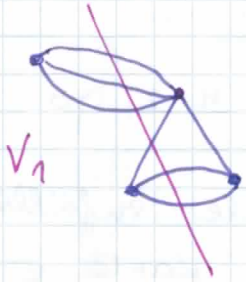
Beispiel:



Definition:

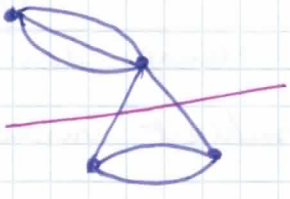
Ein Schnitt (V_1, V_2) $(V_1, V_2 \neq \emptyset)$ ist die Zerlegung der Knotenmenge in zwei Teilen, also so dass $V = V_1 \cup V_2$ und $V_1 \cap V_2 = \emptyset$.

Eine kreuzende Kante (bezüglich (V_1, V_2)) ist eine Kante mit einem Endknoten in V_1 und dem anderen in V_2.



es gibt hier 6 kreuzende Kanten

Ein Schnitt ist minimal, wenn die Anzahl der kreuzenden Kanten minimal ist (unter allen Schnitten von $G(V,E)$).



minimaler Schnitt.

R16.

Beobachtung: Wenn (~~der~~) ein minimaler Schnitt k kreuzende Kanten hat, dann hat jeder Knoten Knotengrad $deg(v) \geq k$.

(Sonst könnte der Knoten alleine eine der Teilmengen V_1 bilden mit weniger als k kreuzenden Kanten)

MIN-CUT Problem

Eingabe: Ein Multigraph $G(V, E)$, $c: E \to \mathbb{N}_0$

Ausgabe: Ein minimaler Schnitt (V_1, V_2)

((mit Hilfe einer vorgegebenen k kann das Problem als Entscheidungsproblem betrachtet werden))

Randomisierter Min-Cut Algorithmus (eine Runde)

Sei $|V| = n$

FOR $i = 1$ to $n-2$ Do

→ wähle zufällig gleichverteilt eine Kante e (parallele Kanten zählen als unterschiedliche Kanten)

→ kontrahiere die Kante e (verschmelze ihre Endknoten)

→ am Ende sind zwei Knoten v_1 und v_2 übrig:
sei V_1 die Menge der Knoten aus V die alle in v_1 kontrahiert wurden, und analog für V_2 und v_2.

→ gib (V_1, V_2) als minimaler Schnitt aus

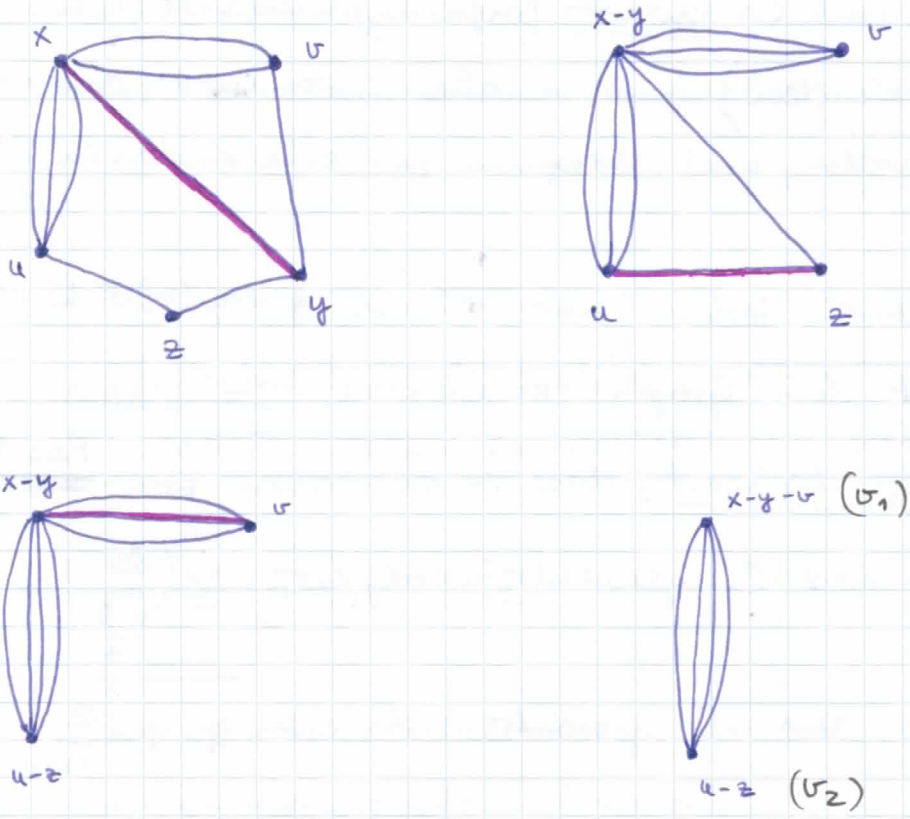
Die Kanten zwischen v_1 und v_2 (nach der $n-2$ Kontraktionen) sind genau die kreuzenden Kanten im Schnitt (V_1, V_2)

Wamm hat die FOR-Schleife n-2 Wiederholungen?

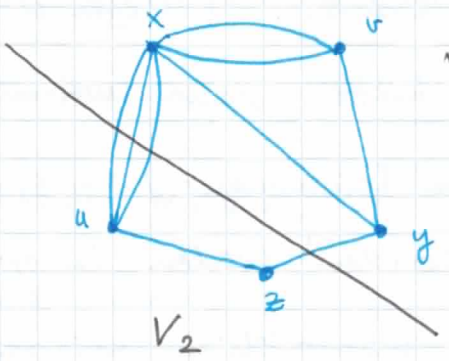
Bei einer Kontraktion von zwei Knoten v und w

- werden v und w verschmelzt (sie zählen als ein Knoten von nun an)
- alle Kanten bleiben erhalten <u>ausser</u> den Kanten zwischen v und w (es entstehen keine Eigenschleifen)

<u>Beispiel für einen Ablauf</u> des Algorithmus:



ergibt den folgenden Schnitt im Eingabegraphen:



Die <u>Reihenfolge</u> der obigen Kontraktionen ist egal! Bitte ausprobieren! Dies gilt auch allgemein.

Wie hoch ist die Fehlerwahrscheinlichkeit des Min-Cut Algorithmus?

R18. Analyse: Für die Analyse fixieren wir einen minimalen Schnitt (W_1, W_2) mit k kreuzenden Kanten. Sei C die Menge der kreuzenden (Einzel-)Kanten $|C| = k$

Wir schätzen die Wahrscheinlichkeit ab (wir geben eine Mindestwahrscheinlichkeit), dass der Algorithmus genau diesen Schnitt (W_1, W_2) (und somit einen minimalen Schnitt) ausgibt.

Dies ist der Fall, wenn der Algorithmus genau die Kanten von C nicht (zufällig) auswählt, (aber alle andere Kanten) und so alle Knoten in W_1 kontrahiert in einen Knoten, und ebenfalls für alle Knoten in W_2.

— Am Anfang hat jeder Knoten v Grad $\deg(v) \geq k$, und deshalb hat der Graph mindestens $\frac{n \cdot k}{2}$ Kanten (Warum?)

Die Wahrscheinlichkeit, dass in der ersten FOR-Runde

→ eine Kante aus C gewählt wird, ist $\leq \frac{k}{\frac{n \cdot k}{2}} = \frac{2}{n}$

→ keine Kante aus C gewählt wird, ist $\geq 1 - \frac{2}{n}$

— Nach der ersten Kontraktion, hat weiterhin jeder der $n-1$ Knoten $\deg(v) \geq k$. (Wenn der kontrahierte Knoten jetzt Grad $< k$ hätte, dann würden seine beiden Originalknoten eine Teilmenge für einen Schnitt mit $< k$ kreuzenden Kanten bilden in G!)

Es gibt somit nach der Kontraktion mindestens $\frac{(n-1) \cdot k}{2}$ Kanten.

Unter der Bedingung, dass in FOR-Runde 1 reine Kante aus C gewählt wurde, die (bedingte) Wahrscheinlichkeit dass in FOR-Runde 2 wieder reine Kante aus C gewählt wird (Erfolgswahrscheinlichkeit in FOR 2), ist

$$\geq 1 - \frac{2}{n-1}$$

⋮

– Angenommen (unter der Bedingung), dass in den ersten $i-1$ FOR-Runden reine Kante aus C gewählt wurde, die (bedingte) Wahrscheinlichkeit, dass in FOR-Runde i wieder reine Kante aus C gewählt wird, ist

$$\geq 1 - \frac{2}{n+1-i}$$

Die Erfolgswahrscheinlichkeit des Min-Cut Algorithmus (alle FOR-Schleifen betrachtet), ist deshalb

$$\geq \left(1-\frac{2}{n}\right)\cdot\left(1-\frac{2}{n-1}\right)\cdot\left(1-\frac{2}{n-2}\right)\cdot \ldots \cdot \left(1-\frac{2}{3}\right) = \frac{2}{n\cdot(n-1)} \geq \frac{2}{n^2}$$

Diese Erfolgswahrscheinlichkeit ist klein! Betrachten wir aber diese wiederholte Version:

Min-Cut $\frac{n^2}{2}$:

– nach $\frac{n^2}{2}$ unabhängigen Durchläufen des Min-Cut Algorithmus, gib von den ausgegebenen $\frac{n^2}{2}$ Ergebnissen den Schnitt (V_1, V_2) mit den wenigsten kreuzenden Kanten aus!

Die Fehlerwahrscheinlichkeit von Min Cut war $\leq 1 - \frac{2}{n^2}$

R 20.

Die Fehlerwahrscheinlichkeit von Min-Cut$_{\frac{n^2}{2}}$ ist

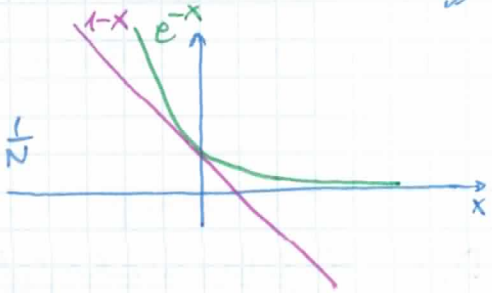
$$\leq \left(1 - \frac{2}{n^2}\right)^{\frac{n^2}{2}} < \frac{1}{e}$$

(Hier wenden wir an, dass $\left(1 - \frac{1}{N}\right)^N < \frac{1}{e}$

((Bekanntlich, $\left(1 - \frac{1}{N}\right)^N \longrightarrow \frac{1}{e}$ falls $N \to \infty$
(von unten)

und allgemein $\left(1 + \frac{a}{N}\right)^N \to e^a$ wenn $N \to \infty$))

„alternativ": $1 - x \leq e^{-x}$ mit $x = \frac{1}{N}$



)

Weitere Wiederholungen von Min-Cut (oder Min Cut$_{\frac{n^2}{2}}$)

zwingen die Fehlerwahrscheinlichkeit unter einen beliebigen Wert.

Beachte, dass $r \cdot \frac{n^2}{2}$, also polynomiell-viele Wiederholungen in n und r, eine Fehlerwahrscheinlichkeit $\frac{1}{e^r}$ (exponentiell klein) erreichen.

Eine randomisierte Datenstruktur:

Skip-Listen (skip-lists)

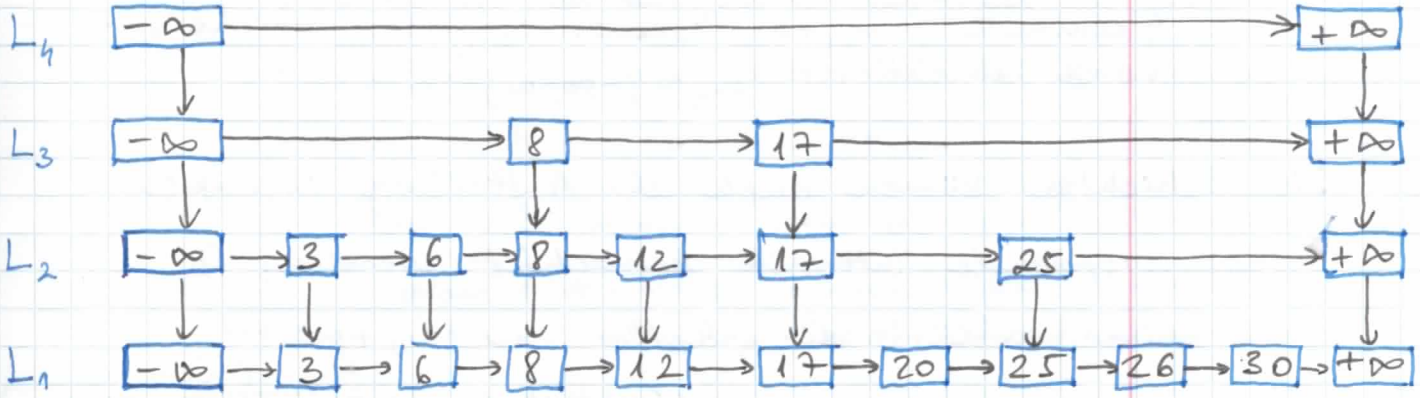
Notation entspricht (Motwani-Raghavan 8.3)

für die Unterstützung der Operationen:

Insert, Delete und Lookup in erwartet logarithmischer Laufzeit

Beispiel: Die Schlüssel $-\infty\{3, 6, 8, 12, 17, 20, 25, 26, 30\}+\infty$

können in einer Skip-Liste z.B. so gespeichert werden (nur eine der vielen Möglichkeiten, weil die Insert-Operation randomisiert ist):



- die Schlüsselmenge wird in Schichten unterteilt

$$L_1 \supseteq L_2 \supseteq L_3 \supseteq \ldots\ldots \supseteq L_t$$

$$L_1 = \{-\infty, 3, 6, 8, 12, 17, 20, 25, 26, 30, +\infty\}$$

$$\vdots$$

$$L_t = \{-\infty, \infty\}$$

- in L_1 werden alle Schlüssel (in einer Liste) gespeichert
- jeder Schlüssel wird mit Prob $= \frac{1}{2}$ auch in Schicht L_2 (in einer Liste) gespeichert, und mit Prob $\frac{1}{2}$ nicht (er wird übersprungen „skipped")

- weiterhin hat jeder Schlüssel in Schicht L_2 einen Zeiger auf die eigene Kopie in L_1

- jeder Schlüssel in L_2 wird mit Prob $= \frac{1}{2}$ auch in der Schicht L_3 (in einer Liste) gespeichert, bzw. mit Prob $= \frac{1}{2}$ nicht mehr gespeichert, übersprungen. Die Reihenfolge der gespeicherten Elementen wird beibehalten. Jeder Schlüssel in Schicht L_3 hat einen Zeiger auf seine Kopie in L_2

⋮

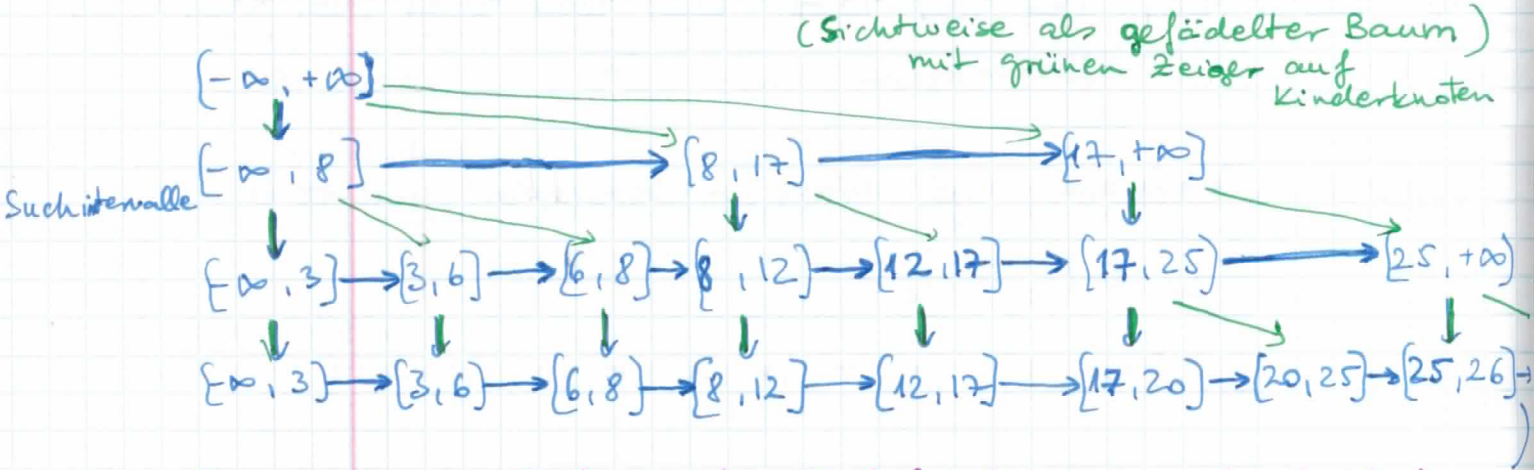
usw.

Bsp. $\ell(8) = 3$

- $\ell(x)$ bezeichne den Index der obersten Schicht von Schlüssel x
- die Elemente $-\infty, +\infty$ kommen in jeder Schicht vor
- die oberste Schicht ist $L_t = \{-\infty, \infty\}$

(Bemerkung:

Die Schichten können auch als Aufteilung in immer gröber werdende Intervalle betrachtet werden. Diese Sichtweise erklärt die Rolle von $-\infty, +\infty$



)

Überlegen wir auf den ersten Blick (anhand des ersten Bildes)

- Schätzen wir den (in Erwartung) benötigten Speicherplatz für n Schlüssel.
- Welche wären die Vorteile bei einer Lookup im Vergleich zu einer gewöhnlichen Liste?

Wie würde man den Schlüssel 26 in der obigen Skip-Liste suchen?

Lookup (x)

→ fange in der Schicht (Liste) L_t an; sei $y = -\infty$

→ REPEAT
- gehe eine Schicht runter zum Kind (Kopie) von y
- finde in der Liste den größten Schlüssel y', der höchstens x ist.
- FALLS $y' = x$ → GEFUNDEN! ELSE setze $y = y'$

UNTIL y in Schicht L_1 (halt)

→ gib NICHT GEFUNDEN! aus

Delete (x)

→ führe Lookup (x) aus: die oberste Schicht L_s die x enthält, wird gefunden ($s = \ell(x)$)

→ entferne x aus den Schichten L_s L_{s-1} L_{s-2} ... L_1 und aktualisiere die Zeiger

(ggf. soll hierfür der Suchpfad - Wenn-x-Nicht-gespeichert - wäre durchlaufen werden)

→ ggf. entferne überflüssige leere Schichten, und reduziere t

Insert (x)

→ ziehe Zufallsbits bis eine 1 gezogen wird; wenn der s-te Versuch die erste 1 ergibt, sei $\ell(x) = s$

→ sei $L_t = \{-\infty, \infty\}$ die bislang höchste Schicht

FALLS $\ell(x) \geq t$, erzeuge neue leere Schichten:

$L_{t+1} = \{-\infty, \infty\}$

$L_{t+2} = \{-\infty, \infty\}$

⋮

$L_{\ell(x)+1} = \{-\infty, \infty\}$

und setze $t = \ell(x) + 1$

→ führe Lookup (x) durch, und ab der Schicht $L_{\ell(x)}$ füge x in die entsprechende Position ein, aktualisiere die Zeiger

Beachte: Die Schicht $\ell(x)$ des Schlüssels wird gemäß der geometrischen Verteilung ausgewürfelt, und bleibt unverändert so lange der Schlüssel gespeichert wird (es werden keinerlei Rotationen oder andere Wartungsarbeiten benötigt, wie es bei balancierten Suchbäumen der Fall ist).

Mit dieser Wahl der Schichten gemäß geometrischer Verteilung wird erreicht, dass die erwartete Anzahl von Schichten und auch mit hoher Wahrscheinlichkeit ihre Anzahl $t = O(\log n)$, wobei n die Anzahl der aktuell gespeicherten Schlüssel ist:

Analyse:

angenommen, unsere Skip-Liste speichert n Schlüssel, und hat aktuell t Schichten

Theorem 1: Die Wahrscheinlichkeit, dass t größer ist als $\alpha \cdot \log_2 n$, ist kleiner als $\frac{1}{n^{\alpha-1}}$ für alle $\alpha > 1$.

Beweis: Seien $x_1, x_2, \ldots, x_n$ die aktuell gespeicherten Schlüssel. $\ell(x_i)$ ist eine Zufallsvariable mit geometrischer Verteilung, unabhängig für jede i und

$$t = \max_i \ell(x_i) + 1$$

$$\text{Prob}\left(\ell(x_i) > 1\right) = \frac{1}{2}$$

$$\text{Prob}\left(\ell(x_i) > 2\right) = \frac{1}{4}$$

$$\text{Prob}\left(\ell(x_i) > 3\right) = \frac{1}{8}$$

$$\vdots$$

$$\text{Prob}\left(\ell(x_i) > r\right) = \frac{1}{2^r} \quad \text{für jede } i \text{ und jede } r$$

Für beliebige r

$\text{Prob}\left(\max_i \ell(x_i) > r\right) =$

$= \text{Prob}\left((\ell(x_1)>r) \text{ oder } (\ell(x_2)>r) \text{ oder } (\ell(x_3)>r) \text{ oder } \dots \text{ oder } (\ell(x_n)>r)\right) \leq$

$\leq n \cdot \text{Prob}\left(\ell(x_i)>r\right) = \frac{n}{2^r}$

↳ weil $\text{Prob}(A \cup B) \leq \text{Prob}(A) + \text{Prob}(B)$ in jedem Fall gilt.

wir setzen $r = \alpha \cdot \log_2 n$

$$\text{Prob}\left(t > \alpha \cdot \log_2 n\right) \simeq \text{Prob}\left(\max_i \ell(x_i) > \alpha \cdot \log_2 n\right) \leq \frac{n}{2^{\alpha \log_2 n}} = \frac{n}{n^\alpha} = \frac{1}{n^{\alpha-1}}$$

□

Insbesondere gilt: $\text{Prob}\left(t > 2 \cdot \log_2 n\right) \leq \frac{1}{n}$

Definition: Wir sagen, dass ein Ereignis mit hoher Wahrscheinlichkeit eintritt (with high probability, w.h.p.), wenn ihre Wahrscheinlichkeit mindestens $1 - \frac{1}{n^\beta}$ ist für irgendein $\beta > 0$, und Eingabelänge n (und für $n > n_0$ für irgendein n_0).

β soll natürlich unabhängig von n sein (für jedes $n > n_0$ gelten)

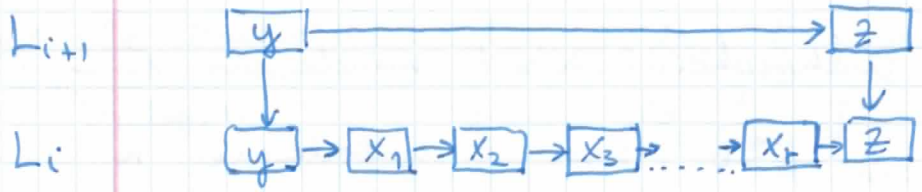
(anders: $\text{Prob} > 1 - \frac{1}{O(n^\beta)}$)

R26.

Als nächstes, schätzen wir die erwartete Laufzeit einer Lookup(x) Operation ab (dies dominiert die Zeit für Insert(x) Delete(x)). Entscheidend hierfür ist die erwartete Länge des Suchpfades von L_t bis zum Schlüssel x (und im Fall von Insert(x) and Delete(x) weiter bis L_1)

- Wir wissen: die Skip-Liste hat mit hoher Wahrscheinlichkeit $O(\log n)$ Schichten.
- Brauchen noch: in den einzelnen Schichten werden jeweils (in Erwartung) nicht zu viele Schlüssel durchlaufen.

Beobachtung: In Schicht L_i wird der Suchpfad von einem Schlüssel y der Schicht L_{i+1} höchstens bis zum nächsten Schlüssel nach y in Schicht L_{i+1} (sei es z) laufen.



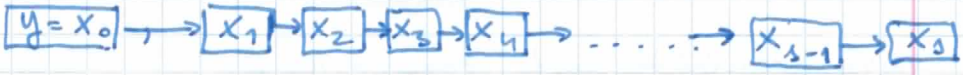
Nennen wir $x_1, x_2, x_3 \ldots x_t$ die Geschwister von y in Schicht L_i (der Name stammt von der Baum-Interpretation). In jeder Schicht wird der Suchpfad nur Geschwister-Schlüssel durchlaufen

Behauptung: Für jede Schicht L_i und jeden Schlüssel $y \in L_i$ die erwartete Anzahl von Geschwistern ist ≤ 1.

Beweis:

→ Wir zeigen die Behauptung nur für die Schicht L_1

→ Seien alle gespeicherten Schlüssel mit Wert mindestens y in (dem Suffix der) Liste L_1 die folgenden:



→ y hat r Geschwister genau dann, wenn

$$\ell(x_1) = \ell(x_2) = \ell(x_3) = \ldots = \ell(x_r) = 1 \quad \text{und} \quad \ell(x_{r+1}) > 1$$

$$\text{Prob}\left(|\text{Geschwister}| = r\right) = \left(\frac{1}{2}\right)^r \cdot \frac{1}{2}$$

(die Anzahl der Geschwister hat (auch) im Wesentlichen geometrische Verteilung → eine geom. Verteilung wo die Werte bei 0 anfangen

$$E\left[|\text{Geschwister}|\right] < \frac{1}{2} \sum_{r=0}^{\infty} r \cdot \left(\frac{1}{2}\right)^r = \frac{1}{2} \cdot E^{\frac{1}{2}}_{\text{geom.}} = \frac{1}{2} \cdot 2 = 1$$

↓ weil $s < \infty$

→ Für höhere Schichten L_i ist der Beweis dasselbe, weil das Argument für beliebige gespeicherte Schlüssel in L_i rechts von y (als Bedingung) gültig ist, und (als bedingten Erwartungswert) ≤ 1 ergibt. □

Theorem 2: Die erwartete Laufzeit von Lookup() ist $\leq 4 \cdot \log_2 n + 1$.

Die erwartete Laufzeit von Insert() und Delete() wird von der Länge eines analogen Suchpfades bestimmt, und ist somit auch $O(\log n)$

R 28.

Beweis:

→ Wenn die Skip-Liste $\leq 2 \cdot \log_2 n$ Schichten hat, ist die erwartete Länge des Suchpfades

$$\leq 2 \cdot \log_2 n \cdot (1+1) = 4 \cdot \log_2 n$$

(1: Schritt nach unten; 1: nach Rechts)

dies geschieht mit Prob $> 1 - \frac{1}{n}$

→ Wenn die Skip-Liste $> 2 \cdot \log_2 n$ Schichten hat, ist die Länge des Suchpfads $\leq n$

$O(n)$ hier wird willkürlich angenommen

($\leq n$ ~~nehmen wir immer an~~)

dies geschieht mit Prob $\leq \frac{1}{n}$

→ die erwartete Laufzeit von Lookup ist deshalb

$$\leq 4 \cdot \log_2 n \left(1 - \frac{1}{n}\right) + n \cdot \frac{1}{n} = 4 \log_2 n + 1$$

(Wir haben hier die ~~Formel~~ für den totalen Erwartungswert benutzt) □.

Theorem 3: Die erwartete Anzahl von Schichten in einer Skip-Liste mit aktuell n Schlüsseln, ist $O(\log n)$.

Beweis: analog wie oben

→ Prob (in Phase 1 oder in Phase 2 ein Paket $\geq 2d$ wartet) $\leq \frac{2}{e^{\frac{d}{2}}}$

→ Plus jedes Paket hat $\leq d$ Schritte pro Phase um die Kanten zu durchlaufen

⇒ mit Prob $\geq 1 - \frac{2}{e^{\frac{d}{2}}}$ ist in $6d$ Schritten jedes Paket am Ziel.

B. FINGERPRINTING (Hromkovič: 1.2 und 4)

Beispiel: Vergleich des Inhalts weit entfernten Datenbanken: Gleichheitstest

→ Ein Computer R_I speichert eine Datenbank (R_I steht irgendwo in Europa); ein anderer Computer R_{II} (in den USA) soll eine Kopie der Datenbank enthalten. Man möchte prüfen ob die beiden Datenbanken identisch sind.

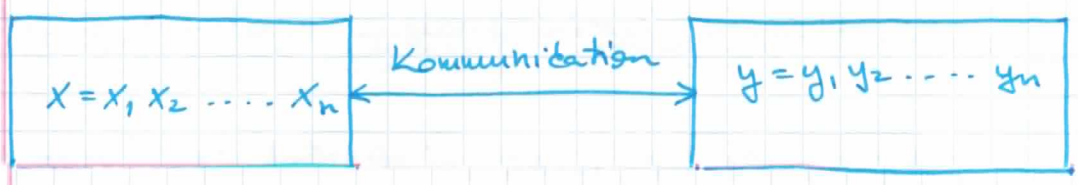
Die Größe der Datenbank ist $n = 10^{16}$ Bits

Ein Kommunikationsprotokoll zwischen R_I und R_{II} soll entscheiden ob die Daten gleich sind. Man möchte die Anzahl der gesendeten Bits niedrig halten.

→ jedes deterministische Protokolle muss mindestens n Bits austauschen zwischen R_I und R_{II} (z.B. einfach die n Bits von R_I an R_{II} schicken)

→ das ist zu viel, und ohne Fehler (Bit-Flip) evtl. gar nicht machbar!

R 40.



Für $x \in \{0,1\}^*$, sei $Num(x) = \sum_{i=1}^{n} 2^{n-i} \cdot x_i$ die Zahl entsprechend der Bitfolge x

Randomisiertes Protokoll für Gleichheitstest

Ausgangssituation: R_I speichert die Bitfolge $x = x_1 x_2 \ldots x_n$
und R_{II} speichert die Bitfolge $y = y_1 y_2 \ldots y_n$

→ R_I wählt zufällig eine Primzahl p aus dem Intervall $[2, n^2]$
(alle Primzahlen aus dem Intervall sind gleichwahrscheinlich)

→ R_I berechnet

$$s = Num(x) \bmod p \text{ , und}$$

schickt s und p an R_{II} (in Binärdarstellung)

→ R_{II} berechnet

$$q = Num(y) \bmod p$$

FALLS $s \neq q$ R_{II} gibt "$x \neq y$" aus
FALLS $s = q$ R_{II} gibt "$x = y$" aus

Analyse der Kommunikationskomplexität

R_I schickt die Binäre Representation von s und p,

für $s \leq p < n^2$

die Länge der Nachricht ist maximal

$2 \cdot \lceil \log_2 n^2 \rceil = 4 \cdot \lceil \log_2 n \rceil$

für $n = 10^{16}$ bedeutet dies

$4 \cdot 16 \cdot \lceil \log_2 10 \rceil = 256$ Bits → kein Problem

Analyse der Fehlerwahrscheinlichkeit

- Die Wahrscheinlichkeit bezieht sich auf die zufällige Wahl von p
- In welcher Richtung kann sich das Protokoll irren?

- falls $x = y$, dann gilt offensichtlich

$$Num(x) \bmod p = Num(y) \bmod p$$

für jede Primzahl p (alle zufällige Wahlen des Protokolls)

in diesem Fall ist die Fehlerwahrscheinlichkeit des Protokolls 0

- falls $x \neq y$, dann kann sich das Protokoll irren,

und zwar genau dann, wenn

$$Num(x) \bmod p = Num(y) \bmod p$$

(Bsp. $x = 0111$ $y = 1010$ p wird zufällig aus

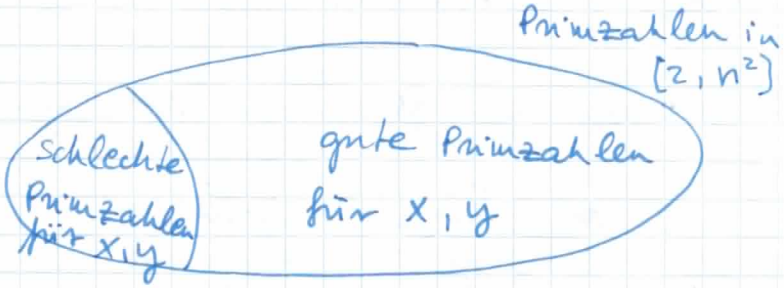
$Num(x) = 15$ $Num(y) = 22$ $\{2, 3, 5, 7, 11, 13, 17, 19, 23\}$ gewählt.

Die Wahl $p = 7$ ergibt falsche Antwort, weil

$15 \bmod 7 = 22 \bmod 7 = 1$, obwohl $x \neq y$)

R42.

Wir zeigen, dass unter allen möglichen $p \in [2, n^2]$ nur relativ wenige eine falsche Antwort liefern falls $x \neq y$



→ Wieviele Primzahlen gibt es insgesamt?

Primzahlensatz: Bezeichne $Prim(m)$ die Anzahl der Primzahlen in $[2, m]$. Es gelten

$$\lim_{m \to \infty} \frac{Prim(m)}{\frac{m}{\ln m}} = 1$$

und $Prim(m) > \frac{m}{\ln m}$ für $m > 67$.

D.h. es gibt ungefähr (und mindestens) $\frac{n^2}{\ln n^2}$ Primzahlen in $[2, n^2]$ für $n \geq 9$

→ Seien x und y fixiert. Höchstens wieviele Primzahlen ergeben falsche Antwort für diese $x \neq y$?

– Die Primzahl p ist schlecht, genau dann wenn

$$Num(x) \bmod p = Num(y) \bmod p$$

d.h. $Num(x) = k \cdot p + s$ und $Num(y) = \ell \cdot p + s$

$$Num(x) - Num(y) = (k - \ell) \cdot p$$

– also, genau dann wenn p teilt $Num(x) - Num(y)$

– Wieviele solche Primzahlen gibt es?

Die Binärdarstellung von x und von y hat jeweils $\leq n$ Bits, deshalb $|Num(x) - Num(y)| < 2^n$

$\Longrightarrow$ es gibt weniger als n verschiedene Primzahlen, die $Num(x) - Num(y)$ teilen, sonst wäre für n verschiedene Primzahlen p_i

$$Num(x) - Num(y) \geq p_1 \cdot p_2 \cdot p_3 \cdot \ldots \cdot p_n > \underbrace{2 \cdot 2 \cdot 2 \cdot \ldots \cdot 2}_{n \text{ mal.}} = 2^n$$

→ Wir erhalten Fehlerwahrscheinlichkeit

$$\leq \frac{n}{\frac{n^2}{\ln n^2}} = \frac{\ln n^2}{n} = 2\,\frac{\ln n}{n}$$

Für $n = 10^{16}$ ist dies $0{,}36892 \cdot 10^{-14}$

Falls jemand diese Fehlerwahrscheinlichkeit für nicht klein genug hält, dann kann sie sehr schnell noch viel niedriger werden:

→ wähle 10 mal hintereinander eine zufällige Primzahl, und gib $x \neq y$ aus wenn $Num(x) \not\equiv Num(y) \bmod p$ für mindestens eine dieser Primzahlen gilt.

Die Fehlerwahrscheinlichkeit:

$$\left(\frac{\ln n^2}{n}\right)^{10} \quad \text{oder allgemein} \quad \left(\frac{\ln n^2}{n}\right)^{k}$$

Für $n = 10^{16}$ ist dies $0{,}4717 \cdot 10^{-141}$

→ alternativ, statt aus $[2, n^2]$ kann eine Primzahl zufällig aus $[2, n^d]$ gezogen werden

So erhalten wir Fehlerwahrscheinlichkeit

$$\leq \frac{n}{\frac{n^d}{\ln n^d}} = \frac{d \cdot \ln n}{n^{d-1}}$$

> In beiden Fällen die Kommunikationskomplexität wächst linear in d (bzw. k), aber die Fehlerwahrscheinlichkeit sinkt mit exponentieller Geschwindigkeit.

Bemerkung: Vergleichen wir jetzt in unserem Beispiel $n = 10^{15}$ die Kommunikation von 2560 Bits für ein Ergebnis mit Fehlerwahrscheinlichkeit $0,4717 \cdot 10^{-141}$ mit der deterministischen Methode, die die Kommunikation von 10^{16} Bits verlangen würde. Nicht nur würde das randomisierte Protokoll viel weniger Kosten verursachen, es wäre auch zuverlässiger: ein Hardware-Fehler und Bit-Flip während der Kommunikation von 10^{16} Bits ist viel Wahrscheinlicher als eine falsche Antwort vom random. Protokoll!

→ In der Praxis, kann ein schneller randomisierter Algorithmus sogar zuverlässiger sein als ein viel langsamer deterministischer Algorithmus!

Warum nennt man diese Methode Fingerprinting?

→ große Objekte, bzw. ihre String Representationen x und y sollen auf Gleichheit geprüft werden. Die vollen String Representationen sind viel zu groß und können nicht vollständig verglichen werden...

→ Deshalb werden stattdessen nur ihre Fingerabdrücke h(x) und h(y) verglichen. Diese Fingerabdrücke sind kürzer, und können deshalb schnell verglichen werden. Wenn die Fingerabdrücke unterschiedlich sind, dann sind die Objekte auch unterschiedlich

$$(h(x) \neq h(y)) \Longrightarrow (x \neq y)$$

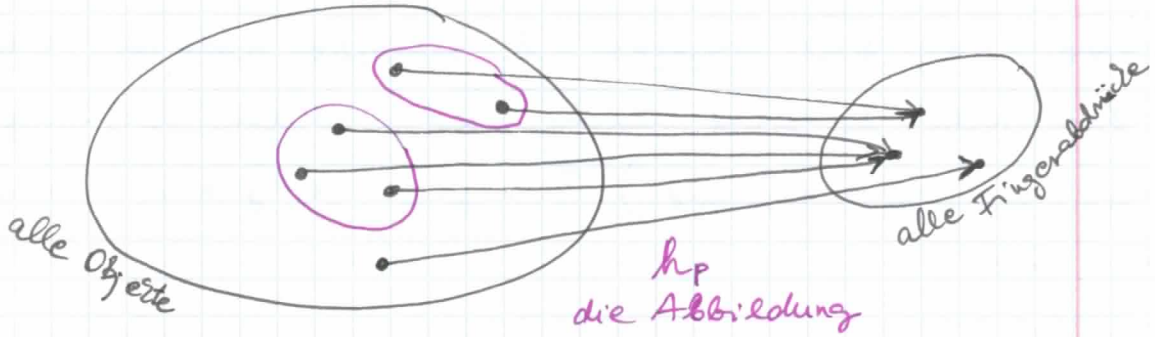
In unserem Beispiel waren x und y lange Strings, und

$$h(x) = Num(x) \bmod p$$

$$h(y) = Num(y) \bmod p$$

die Fingerabdrücke.

→ Die Fingerabdrücke müssen kürzer sein als die zu vergleichende Objekte x und y; sie sind somit notwendigerweise unvollständige Darstellungen von x und y. Anders gesagt: es gibt deutlich weniger mögliche Fingerabdrücke als mögliche Objekte: es müssen Objekte geben, die den selben Fingerabdruck besitzen.



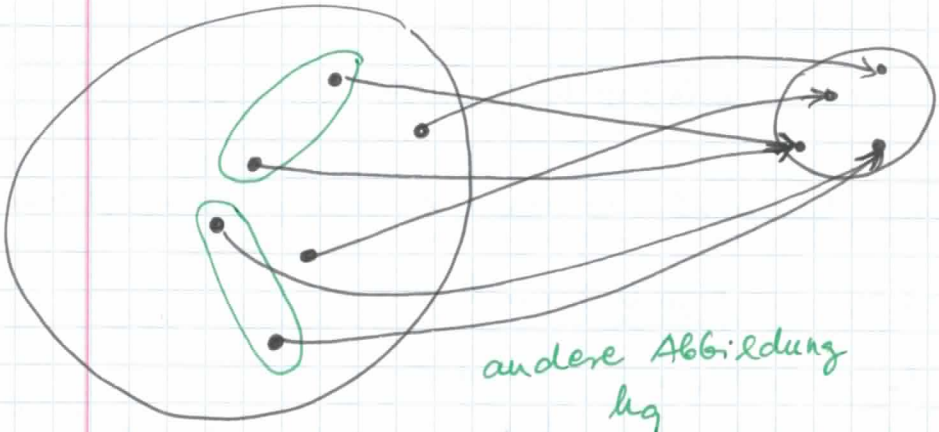
es kommt also vor, dass $x \neq y$ aber $h(x) = h(y)$

→ Was nun?

Eigentlich definiert in unserem Beispiel jede einzelne Primzahl aus $[2, n^2]$ eine eigene solche Abbildung

$$h_p(x) = Num(x) \bmod p$$

Wir haben also mit einer ganzen Menge M von Abbildungen zu tun (eine pro Primzahl)



eine andere Abbildung h_q aus M bildet ganz andere Objekt-Mengen auf denselben Fingerabdruck als h_p

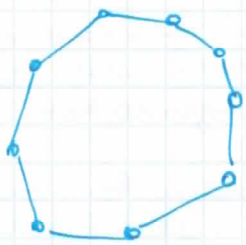
→ Die Menge von Abbildungen M soll so gewählt werden, dass für jedes $x \neq y$ Paar, die meisten Abbildungen $h \in M$ gut sind, d.h. $h(x) \neq h(y)$. Dann wird eine zufällig gewählte h (im Beispiel zufällige h_p) sehr wahrscheinlich $h(x) \neq h(y)$ und $x \neq y$ ausgeben.

(Für jede $x \neq y$ sind manche h schlecht aber die meisten gut; für jede h sind manche $x \neq y$ so dass $h(x) = h(y)$, aber die meisten $x \neq y$ gut) Für jede $x \neq y$ sind andere $h \in M$ gut.

C. SYMMETRY-BREAKING

- In fast homogenen Strukturen (Mengen, Graphen) werden den Elementen / Knoten .. Rollen, Reihenfolgen, Prioritäten zugewiesen, indem sich jedes Element zufällig eine Rolle auswählt.
- Es kommt eher in parallelen oder verteilten Systemen vor: die Prozessoren werden in unterschiedliche Typen aufgeteilt, mit jeweils anderen Aufgaben o. Zielen
- und/oder: unter vielen symmetrischen Lösungen wird so eine Lösung ausgewählt

z.B. maximale unabhängige Knotenmenge in diesem Graphen



R 48.

Wir betrachten parallele Graph-Algorithmen für das folgende Maschinenmodell:

Shared-Memory Architekturen (PRAM-Modell)

→ es gibt Prozessoren (RAMs), jeder mit einem kleinen lokalen Speicher

→ sie haben Zugriff auf einen gemeinsamen globalen Speicher: jeder Prozessor kann jedes Register lesen oder beschreiben

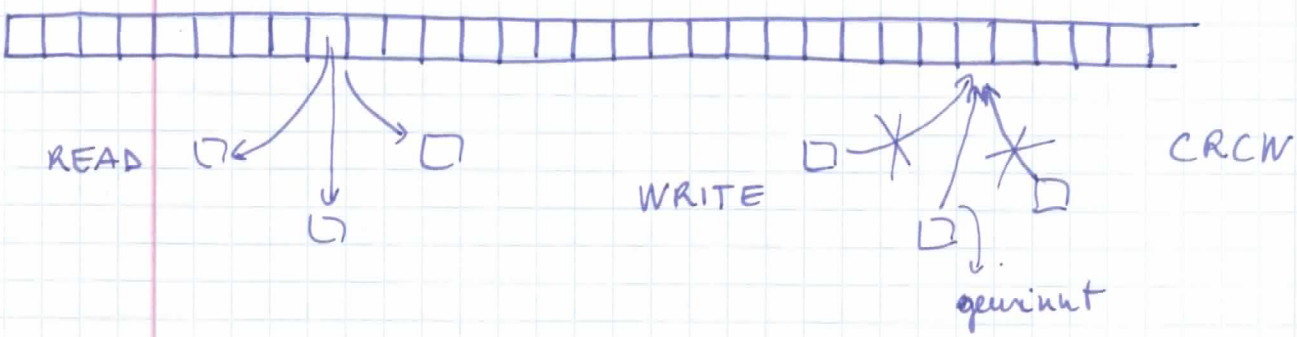
→ die Berechnung wird in synchronisierten parallelen Schritten durchgeführt; in einem parallelen Schritt darf jeder Prozessor

- ein globales Register lesen
- lokale Operationen durchführen
- versuchen ein globales Register zu beschreiben

– wenn mehrere Prozessoren dasselbe Register beschreiben wollen, gewinnt irgendeiner von ihnen.

– sie können dasselbe Register lesen

– die Prozessoren kommunizieren ausschließlich über dem globalen Speicher miteinander



Beispiel für Symmetry-Breaking:

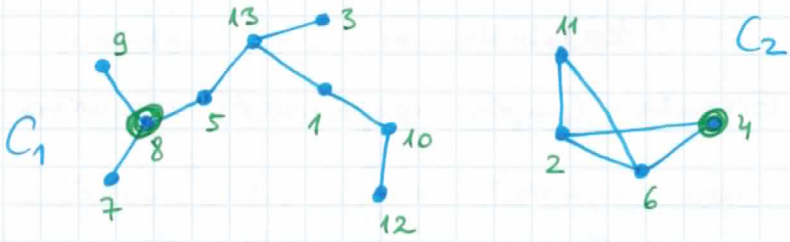
Zusammenhangskomponenten

Eingabe: ein ungerichteter Graph $G(V, E)$ mit
n Knoten und m Kanten

Aufgabe: die Bestimmung aller Zusammenhangskomponenten

Was bedeutet „Zusammenhangskomponente"?

Beispiel:



hat 2 Zusammenhangskomponenten, C_1 und C_2

(Def: v und w sind genau dann in der selben Zusammenhangskomponente C_i, wenn es einen Weg von v nach w im G gibt)

– Wie werden in der Lösung die Zusammenhangskomponenten repräsentiert?

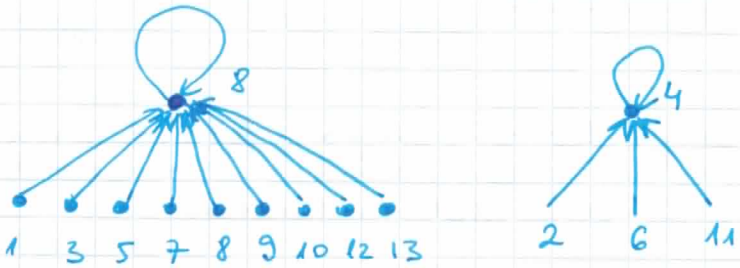
Schließlich wird in jeder Komponente ein Knoten ausgezeichnet, und in einem n-elementigen Array Vater[ ] wird für jeden Knoten der ausgezeichnete (Vater) Knoten aus seiner Komponente, gespeichert

Im obigen Beispiel:

Vater[i] [8, 4, 8, 4, 8, 4, 8, 8, 8, 8, 4, 8, 8]

i	1	2	3	4	5	6	7	8	9	10	11	12	13

anders gesagt: schließlich wird jeder Zusammenhangs-Komponente ein solcher Stem entsprechen:



die Wurzel ist der ausgezeichnete Knoten; sie hat eine Eigenschleife Vater[i] = i

die anderen Knoten zeigen auf ihren Vater-Knoten

— Wie wird der Eingabe-Graph gespeichert, wie wird parallelisiert?

Wir nehmen einen Prozessor für jeden Knoten, und einen für jede Kante, also n+m Prozessoren.
Jeder Kanten-Prozessor kennt seine zwei Endknoten.
(In der Praxis kann es sich hier um virtuelle Prozessoren handeln.)

— Der Eingabegraph und der Vater-Array werden global gespeichert.

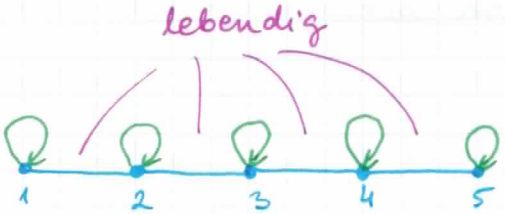
Die Idee eines schnellen parallelen randomisierten Algorithmus

Notation: Graph-Kante ——●

Stem-Kante ——→

- der Alg. fängt mit einzelnen (trivialen) Sternen an

Runde 0



Vater = [1,2,3,4,5]

- wenn ein Kanten-Prozessor $\{u,v\}$ sieht, dass seine Endknoten zu unterschiedlichen Sternen gehören (solche Kanten heißen lebendig), versucht er die beiden Sterne zu verschmelzen

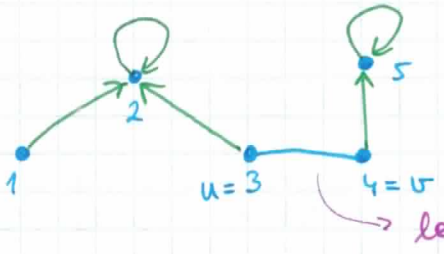
- Wie werden sie verschmelzt?

Vater[Vater[u]] ← Vater[v] verschmelzt die zwei Wurzel

für jeden Knoten i

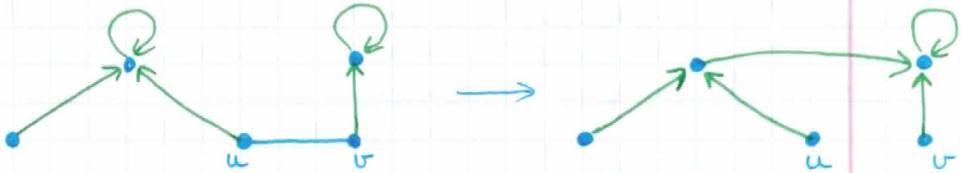
Vater[i] ← Vater[Vater[i]] stellt dann die Zeiger der Satelliten um

Runde 1



Vater = [2,2,2,5,5]

Runde 2



Vater[Vater[u]] = Vater[v]

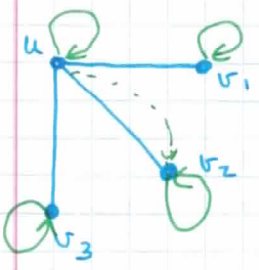
for i = 1 to n

Vater[i] = Vater[Vater[i]]

R52.

Es bleiben noch offene Fragen:

1. Was passiert in der folgenden Situation?

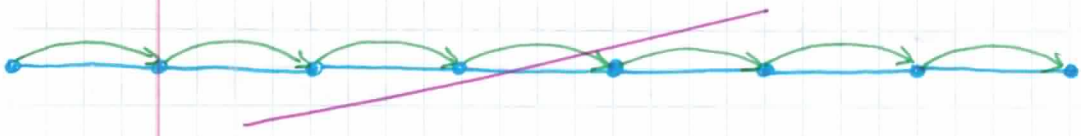


die drei Kanten versuchen parallel v_1 und v_2 und v_3 als Vater [u] einzutragen

→ nicht schlimm! Irgendeine Kante (egal welche) gewinnt

2.

Wäre so eine Runde erlaubt?



NEIN!

Um Fall 2. zu verhindern, brechen wir die Symmetrie:

am Anfang jeder Runde wählt jeder Stern zufällig ein Geschlecht, und wird mit Wahrscheinlichkeit $\frac{1}{2}$ männlich, bzw. weiblich.

(Das Geschlecht wird von der Wurzel zufällig gewählt, und dann parallel von allen Kindern übernommen.)

nur lebendige Kanten zwischen unterschiedlichen (einem männlichen und einem weiblichen) Sternen versuchen die Sterne zu verschmelzen, und zwar nur in Richtung der weiblichen Wurzel: die Kante versucht die weibliche Wurzel in das Register (Vater-Array Eintrag) der männlichen zu schreiben.

→ Somit werden Ketten von Verschmelzungsschritten verhindert.

Der Algorithmus:

1. for $i = 1$ to n pardo $Vater[i] = i$ (Initialisierung)

2. WHILE es lebendige Kanten gibt, DO

   - jede Wurzel w (mit $Vater[w] = w$) wählt ein Geschlecht aus $\{0, 1\}$ zufällig, und die Kinder von w übernehmen dieses Geschlecht

   O(1) parallel

   - wenn u zu einem männlichen Stern mit Wurzel w_0, und v zu einem weiblichen mit Wurzel w_1 gehört, dann versucht der Kanten-Prozessor $\{u, v\}$ w_1 in das Register von w_0 zu schreiben ($\underline{\text{versucht}}$ $Vater[w_0] := w_1$)
     ingendeine Kante inzident mit dem Stern von w_0 – wenn eine gibt – wird gewinnen

   O(1) parallel

   - for $i = 1$ to n pardo $Vater[i] = Vater[Vater[i]]$

   O(1) parallel

R54.

Korrektheit: - Die Knoten eines jeden Stems gehören in jeder Runde zur selben Zusammenhangskomponente
- Die Sterne werden so lange verschmelzt, bis es Kanten zwischen verschiedenen Sternen (lebendige Kanten) gibt.

$\Rightarrow$ alle Zusammenhangskomponenten werden gefunden.

Laufzeit: Jede Iteration der WHILE-Schleife läuft in paralleler $O(1)$ Zeit.

Wieviele Iterationen gibt es?

Theorem: Mit hoher Wahrscheinlichkeit $\left(p > 1 - \frac{1}{n}\right)$ genügen $5 \log n$ Iterationen der While-Schleife.

Beweis: Ein Knoten heißt lebendig, wenn er die Wurzel eines lebendigen Stems ist (der noch keine ganze Zusammenhangskomponente ist). Am Anfang sind alle Knoten lebendig

Beobachtung: Ein lebendiger Knoten verschwindet in einer beliebigen Iteration mit Wahrscheinlichkeit $\geq \frac{1}{4}$.

Sei ein lebendiger Knoten w die Wurzel des Stems S, und sei $\{u, v\}$ eine lebendige Kante zwischen S und einem anderen Stem S'.
Mit Wahrscheinlichkeit $\frac{1}{4}$ wird S männlich und S' weiblich, und in diesem Fall verschwindet w weil er an S' oder an einen anderen Stem angehängt wird.

Sei v ein fixierter Knoten. Die Wahrscheinlichkeit, dass v nach $5\log_2 n$ Iterationen lebendig ist, ist

$$p_v \leq \left(1-\frac{1}{4}\right)^{5\cdot\log_2 n} = \left(\frac{3}{4}\right)^{5\cdot\log_2 n} = \left(\frac{243}{1024}\right)^{\log_2 n} \leq \left(\frac{1}{4}\right)^{\log_2 n} = \frac{1}{n^2}$$

$$P(\text{irgendein Knoten in } V \text{ nicht verschwindet nach } 5\cdot\log_2 n \text{ Runden}) \leq$$

$$\leq \sum_{v\in V} P(v \text{ nicht verschwindet}) = n\cdot p_v \leq n\cdot\frac{1}{n^2} = \frac{1}{n} \qquad \square$$

Zum Thema FINGERPRINTING:

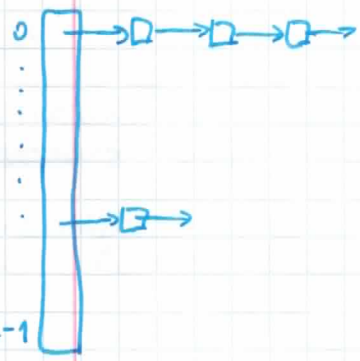
Universelles Hashing

Zur Erinnerung: Hashing mit Verkettung

- ein grosses Universum U von möglichen Schlüsseln ist gegeben $|U| = p$

- die aktuellen Schlüssel werden in einem Array von Listen (einer Tabelle) gespeichert. Die Operationen insert(), remove() und lookup() werden unterstützt

die Tabelle hat m Zellen



- eine Hash-Funktion

$$h: U \longrightarrow \{0, 1, \ldots, m-1\}$$

bestimmt die Zelle (bzw. die Liste) wo ein gegebener Schlüssel $x \in U$ gespeichert wird.

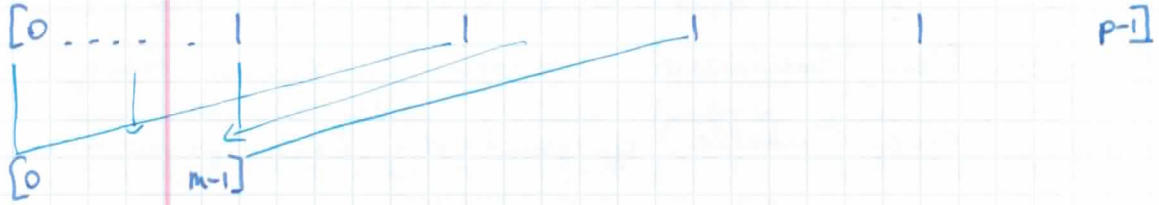
- da $m << |U|$ gilt, wird unvermeidlich $h(x) = h(y)$ für manche $x \neq y \in U$ gelten.

- in einer Folge von n Insert-Operationen können im Worst-Case alle Schlüssel in die selbe Zelle gehasht werden; dies ergibt im Worst Case eine Liste der Länge n, und $\Theta(n^2)$ Laufzeit für die n Operationen

Andererseits ist die Anzahl <u>gespeicherter</u> Schlüssel fast immer $< m$, also wir haben gute Chancen...

- Sei $\mathcal{U} = \{0, 1, 2, \ldots, p-1\}$ für eine Primzahl (dies können wir o.B.d.A. annehmen)

<u>Naheliegende Wahlen für Hash-Funktionen</u>



- $h(x) = x \mod m$

- die Schlüsselmenge kann um einen beliebigen Wert b zyklisch auf $[0 \ldots p-1]$ verschoben werden und dann (mod m) gehasht werden:

$$h(x) = ((x+b) \mod p) \mod m$$

eine Permutation

- Die Schlüssel können zuerst permutiert werden auf $[0, 1, \ldots p-1]$ mit Hilfe einer mod p Multiplikation, und dann mod m gehasht werden

$$h(x) = ((a \cdot x) \mod p) \mod m$$

$f(x) = a \cdot x \mod p$ ist eine Permutation falls p Prim und $a \neq 0$

Beispiel: Was passiert mit $U = \{0, 1, 2, \ldots 6\}$
(d.h. für $p = 7$) bei der Multiplikation

$$f(x) = 3 \cdot x \mod 7$$

x:	$f(x)$
0	0
1	3
2	6
3	2
4	5
5	1
6	4

(hier sieht man, dass zB. 5 und 3 multiplikative Inversen füreinander sind mod 7 (in $\mathbb{F}_7$)
d.h. $5 \cdot 3 \equiv 1 \quad (7)$
$(5)^{-1} = 3 \quad (7)$

– wir können beide Abbildungen kombinieren allgemein in Form einer linearen Funktion:

$$h(x) = ((ax + b) \mod p) \mod m$$

$(ax + b) \mod p$ permutiert (falls $a \neq 0$) und mod m „hasht"

(es gibt viele andere Hash-Funktionen)

R89. **Universelles Hashing**

Für jede einzelne Hash-Funktion können wir aber mit der Eingabe Pech haben (bzw. einen bösartigen Gegner haben der die Eingabe generiert), und lange Listen in der Hash-Tabelle erhalten.

Deshalb wählen wir unsere Hash-Funktion unter allen $h_{a,b}$ zufällig aus. Dann wird $h_{a,b}$ in Erwartung für unsere Eingabe gut.

($h_{a,b}$ wählen wir einmal, vor allen Operationen aus; der Gegner kennt unsere $h_{a,b}$ nicht (oblivious adversary), nur die Menge der möglichen Hash-Funktionen (genauer: die Verteilung), und so wählt er die Operationen-Folge)

Eine Hash-Funktion $h_{a,b}$ wird zufällig gleichverteilt aus der Menge

$$H = \{ h_{a,b} \mid 1 \leq a \leq p-1 \;,\; 0 \leq b \leq p-1 \} \text{ ausgewählt}$$

wobei $h_{a,b}(x) = (ax + b \bmod p) \bmod m$

Was ist $|H|$? $|H| = p \cdot (p-1)$

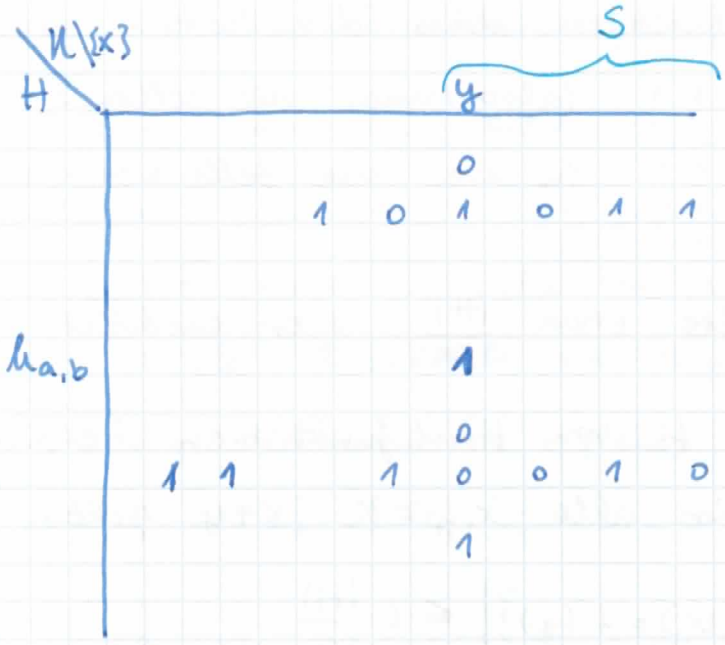
Intuitiv, die Menge H ist gut, wenn die Funktionen $h_{a,b}$ nicht immer die selbe Gruppe von Schlüsseln in die selbe Zelle hashen. (Dies könnte passieren, auch wenn jede einzelne $h_{a,b}$ die Schlüssel gleichmäßig verteilt.) Als Nächstes, formalisieren wir diese Forderung präziser:

Sei $x \in U$ ein fixierter Schlüssel

Zur Illustration definieren wir eine Matrix, wo die Zeilen den Hash-Funktionen, und die Spalten allen Schlüsseln y entsprechen (ausser dem Schlüssel x).

der Eintrag 1 bedeutet: $h_{a,b}(y) = h_{a,b}(x)$

der Eintrag 0 bedeutet $h_{a,b}(y) \neq h_{a,b}(x)$



Beobachtung 1. Da $m \ll p$, muss jede Hashfunktion $h_{a,b}$ durchschnittlich $\sim \frac{p}{m}$ Schlüssel in die selbe Zelle $h_{a,b}(x)$ hashen, also in jeder Zeile sollten $\sim \frac{p}{m}$ 1-er sein, zumindest für Hash-Funktionen, die die Schlüssel gleichmäßig verteilen.

(wir haben gesehen, dass alle $h_{a,b}$ solche sind, weil $(ax+b \bmod p)$ permutiert, und $(\bmod m)$ gleichmäßig verteilt)

Beobachtung 2. Es gibt also etwa $\frac{p}{m} \cdot |H|$ 1-er in der Tabelle, weil es $|H|$ Zeilen gibt.

Beobachtung 3. Die Spalten haben durchschnittlich

$$\frac{p}{m}|H| \Big/ p = \frac{|H|}{m} \quad 1\text{-er.}$$

Unsere Forderung: wir möchten, dass die Matrix nicht so aussieht: (also immer die selben Schlüssel in die selbe Zelle wie x),

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

sondern dass jede Spalte etwa $\frac{|H|}{m}$ 1-er enthält.

Definition: Eine Menge H von Hashfunktionen heißt c-universell, falls für alle $x, y \in N$ $x \neq y$ gilt:

$$\text{„Anzahl der 1-er in Spalte } y\text{“} = \left|\{h \in H \mid h(x) = h(y)\}\right| \leq c \cdot \frac{|H|}{m}$$

Theorem: Die Menge von Hashfunktionen

$$H = \{h_{a,b} \mid 1 \leq a \leq p-1,\ 0 \leq b \leq p-1\}$$

ist c-universell für $c = \left(\left\lceil \frac{p}{m} \right\rceil \Big/ \frac{p}{m}\right)^2 \approx 1$

[Beweis: → Seien $x, y \in N$ beliebig und fixiert.
Wir zählen die $h_{a,b} \in H$ für die $h_{a,b}(x) = h_{a,b}(y)$ gilt.
→ Es gilt wenn $(ax + b \bmod p) \bmod m = (ay + b \bmod p) \bmod m$
Was bedeutet das?

Wenn wir $ax+b \bmod p$ und $ay+b \bmod p$ durch m teilen, der Rest ist dasselbe, sei er q

$$ax + b \bmod p = q + r \cdot m$$
$$ay + b \bmod p = q + s \cdot m \quad \text{für irgendein } q, r, s$$

(Man braucht zu wissen: wenn p eine Primzahl ist, dann ist die Zahlenmenge $\{0, 1, 2, \dots, p-1\}$

so, dass man mit Addieren, Subtrahieren, Multiplizieren Dividieren modulo p mit diesen Zahlen „alles machen kann", was man mit den Zahlen in $\mathbb{R}$ oder $\mathbb{Q}$ machen kann. Viele Resultate der Algebra „bleiben" gültig in diesem sog. Zahlenkörper (field) namens $\mathbb{F}_p$.

Beispiel: in $\mathbb{F}_7$ gilt $\frac{2}{3} = 3$, weil $3 \cdot 3 \equiv 2 \bmod 7$)

also über dem Körper $\mathbb{F}_p$ gilt

$$ax + b = q + r \cdot m$$
$$ay + b = q + s \cdot m$$

wobei $q \in \{0, 1, \dots m-1\}$
und $r, s \in \{0, 1, \dots, \lceil \frac{p}{m} \rceil - 1\}$

und (a, b) bestimmen (r, s, q) eindeutig (s.d. $q \in \{0, 1, \dots, m-1\}$).

<u>Andersrum auch</u>: jede solche (r, s, q) bestimmt die Zahlen

$z = q + r \cdot m < p$ und $w = q + s \cdot m < p$ und

das Gleichungssystem

$$ax + b = z$$
$$ay + b = w \qquad z \neq w$$

hat eine eindeutige Lösung (a, b)

→ x und y sind fixiert;

es gibt $m \cdot \lceil \frac{p}{m} \rceil \left(\lceil \frac{p}{m} \rceil - 1 \right)$ solche (q, r, s) dass $r \neq s$, und zu jedem Tupel (q, r, s) ein (a, b) Paar, so dass $h_{a,b}(x) = h_{a,b}(y)$

→ Die Anzahl der $h_{a,b}()$ Funktionen s.d. $h_{a,b}(x) = h_{a,b}(y)$ ist somit

$$\approx m \cdot \lceil \frac{p}{m} \rceil^2 \approx \frac{p^2}{m} \approx \frac{|H|}{m}$$

⇒ H ist c-universell mit $c \approx 1$. □]

Analyse der Laufzeit:

→ Angenommen, eine beliebige Folge von n-1 Insert, Remove und Lookup Operationen ist schon hinter uns. Wir betrachten die erwartete Laufzeit der n-ten Operation Insert(x) oder Remove(x) oder Lookup(x)

→ Die Erwartung ist über der Wahl von $h_{a,b}$ am Anfang, die gespeicherten Schlüssel und x sind beliebig gewählt. Den gesuchten Erwartungswert bezeichne E_n.
Die n-te Operation sei oBdA Insert(x).

→ Sei S die Schlüsselmenge gespeichert vor der n-ten Op. und sei K_n (Kollision) die erwartete Anzahl der Schlüssel in Zelle h(x) (Länge der Liste)
Dann gilt $|S| \leq n-1$ und $E_n = K_n + 1$

für fixierte h das sind die 1-er in den Spalten von S innerhalb der Zeile h

$$K_h = \sum_{h \in H} \frac{1}{|H|} \left| \{ y \in S \mid h(x) = h(y) \} \right| =$$

Zeile für Zeile summieren wir die Anzahl der 1 in den Spalten von S

$$= \frac{1}{|H|} \cdot \sum_{h \in H} \sum_{\substack{\{y \in S \mid \\ h(x) = h(y)\}}} 1 =$$

Spalte für Spalte summieren wir die Anzahl der 1

$$= \frac{1}{|H|} \sum_{y \in S} \sum_{\substack{\{h \in H \mid \\ h(x) = h(y)\}}} 1 =$$

weil H c-universell

$$= \frac{1}{|H|} \sum_{y \in S} \left| \{ h \mid h(x) = h(y) \} \right| \leq \frac{1}{|H|} \sum_{y \in S} c \cdot \frac{|H|}{m} =$$

$c \cdot \lambda$ wenn Auslastungsfaktor λ nicht überschritten wird

$$\leq c \cdot \frac{|S|}{m} \leq c \cdot \frac{n-1}{m}$$

Es folgt: $E_h = 1 + K_h \leq 1 + c \cdot \frac{n-1}{m}$

→ für unsere H gilt $c = \left(\frac{\lceil \frac{p}{m} \rceil}{\frac{p}{m}} \right)^2 \approx 1$ für $p \gg m$

→ $1 + c \cdot \frac{|S|}{m}$ ist für Auslastungsfaktor $\frac{|S|}{m}$ praktisch optimal

$$E_1 + E_2 + \ldots + E_n \leq \sum_{i=1}^{n} \left(1 + c \cdot \frac{i-1}{m}\right) \approx n \left(1 + \frac{c}{2} \cdot \frac{n}{m}\right) \leq n(1+c\lambda)$$

für n Operationen