Grundzüge der Algorithmen und Datenstrukturen

Skript zur Vorlesung von Prof. Wolfgang Paul
für das Wintersemester 2018

Letzte Änderung: 23. Februar 2019

— Öffentliche Version —

Autoren
(in chronologischer Reihenfolge)

Jannis Köhl
Philippe Heim
Lena Becker
Hanna Nebelung
Felix Jahn
Julia Laichner
Julia Tillman
Timon Ulrich
Jessica Schmidt
Niklas Medinger
Jule Enninghorst
Simon Schwarz
Daniel Weber
Bastian Herra

Inhaltsverzeichnis

1 Effiziente Multiplikation

1.1 Römische Tabelle

Es gilt

$$(a+b)^2 = a^2 + 2ab + b^2$$

sodass aus der Formel

$$ab = \frac{(a+b)^2 - a^2 - b^2}{2}$$

folgt, dass eine Tabelle, welche nur Quadratzahlen enthält, ausreicht um jedes beliebige Produkt zu berechnen. Im Folgenden wird ein weiteres Verfahren zur Optimierung von Multiplikation vorgestellt.

1.2 Hilfsschaltkreise

1.2.1 Vektor-$\wedge$

Seien $x \in \mathbb{B}$ und $v \in \mathbb{B}^n$. Definiere $x \wedge v = (x \wedge v_{n-1}) \dots (x \wedge v_0)$, sodass

$$x \cdot \langle a \rangle = \langle x \wedge a \rangle = \begin{cases} a & \text{falls } x = 1 \\ 0 & \text{falls } x = 0 \end{cases}$$

mit Kosten von $\mathcal{O}(n)$.

1.2.2 Potenz

Sei $x \in \mathbb{B}$. Definiere $x^n = x \cdots x$ (n-Mal) mit $\mathcal{O}(1)$.

1.2.3 High-Low-Split

Sei $a = a_{n-1} \cdots a_0$. Dann gilt $\langle a \rangle = \langle a_{n-1} \cdots a_k \rangle \, 2^k + \langle a_{k-1} \cdots a_0 \rangle$ mit $\mathcal{O}(1)$.

1.3 Algorithmus von Karazuba

1.3.1 Notation

Manchmal schreiben wir einfach x an Stelle von $\langle x \rangle$.

1.3.2 Idee

Seien $x, y \in \mathbb{B}^{2k}$ und $n = 2k$. Es existieren $a, b, c, d \in \mathbb{B}^k$ mit $x = a2^k + b$ und $y = c2^k + d$. Definiere

$$u = ac \qquad v = bd$$
$$w = (a+b)(c+d) - u - v = ad + bc$$

sodass

$$xy = (a2^k + b)(c2^k + d) = ac2^{2k} + (ad + bc)2^k + bd = u2^n + w2^k + v \qquad (\star)$$

mit nur 3 statt 4 Multiplikationen berechnet werden kann.

1.3.3 Kosten

Wir führen die folgenden Schaltkreiskosten ein

Operation	Zeichen
Multiplikation	$M(n)$
Addition	$A(n)$
Subtraktion	$S(n)$

und bezeichnen die Kosten für Ausdruck x mit $C(x)$. Damit gilt

$$C(u) = C(v) = M(k)$$
$$C(w) = 2A(k) + M(k+1) + 2S(n)$$

Es ist $M(k+1)$, da $a+b$ und $c+d$ höchstens $k+1$ Bit haben. Nach $(\star)$ benötigen wir für die Berechnung von xy aus u, v, w zusätzlich $2A(2n)$.

Wir versuchen nun, $M(k+1)$ zu $M(k)$ zu vereinfachen. Es existieren $e_k, f_k \in \mathbb{B}$ und $e, f \in B^k$ mit $a + b = \langle e_k e \rangle$ und $c + d = \langle f_k f \rangle$. Somit gilt

$$(a+b)(c+d) = (e_k 2^k + e)(f_k 2^k + f) = f_k e_k 2^n + e_k f 2^k + e f_k 2^k + ef$$

Das Aufteilen von $a+b$ und $c+d$ ist kostenlos, $f_k e_k$ ist ein einfaches „und“, $e_k f$ ist $e_k \wedge f$ (mit Kosten k), ef_k ist $e \wedge f_k$ (mit Kosten k) und ef kostet $M(k)$. Wir folgern

$$M(k+1) \leq M(k) + 2k + 1$$

Also gilt

$$C(w) = 2A(k) + M(k+1) + 2S(n) \leq 2A(k) + M(k) + 2S(n) + 2k + 1$$

Mit dem Wissen, dass $A(n) = S(n) = \mathcal{O}(n)$ gilt, folgt

$$\begin{aligned} M(n) &= C(xy) \\ &= C(u) + C(v) + C(w) + 2A(2n) \\ &\leq 3M(k) + 2A(2n) + 2A(k) + 2S(n) + 2k + 1 \\ &= 3M(k) + \mathcal{O}(n) \end{aligned}$$

Nehmen wir an, die Ungleichung ist eine Gleichung. Dann gilt nach dem Master-Theorem

$$M(n) = 3M(k) + \mathcal{O}(n) \quad M(1) = \mathcal{O}(1)$$
$$\Rightarrow M(n) = \mathcal{O}(n^{\log 3}) \approx \mathcal{O}(n^{1.59})$$

2 Master-Theorem

2.1 Aussage

Sei $T : \mathbb{N} \to \mathbb{R}$ gegeben mit $a \in \mathbb{N}$ und

$$T(n) = \begin{cases} b & n = 1 \\ aT(n/2) + bn & \text{sonst} \end{cases}$$

Dann gilt

$$T(n) = \begin{cases} \mathcal{O}(n) & a = 1 \\ \mathcal{O}(n \log n) & a = 2 \\ \mathcal{O}(n^{\log a}) & a \geq 3 \end{cases}$$

2.2 Beweis

2.2.1 Geometrische Reihe

Sei $c \in \mathbb{R}$ beliebig.

$$\begin{aligned} s &= \sum_{i=0}^{n-1} c^i \\ \Rightarrow cs &= \sum_{i=1}^{n} c^i \\ \Rightarrow cs - s &= \sum_{i=1}^{n} c^i - \sum_{i=0}^{n-1} c^i = c^n - 1 \\ \Rightarrow s &= \frac{c^n - 1}{c - 1} \end{aligned}$$

2.2.2 Hauptteil

$$
\begin{aligned}
a = 1: \qquad T(n) &= T(n/2) + bn \\
&\leq b + bn \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i \\
&= \mathcal{O}(n)
\end{aligned}
$$

$$
\begin{aligned}
a = 2: \qquad T(n) &= 2T(n/2) + bn \\
&= 2^{\log n} b + \sum_{i=0}^{\log(n)-1} 2^i b \frac{n}{2^i} \\
&= bn^{\log 2} + bn \sum_{i=0}^{\log(n)-1} 1^i \\
&= bn^{\log 2} + bn \log(n) \\
&= \mathcal{O}(n \log n)
\end{aligned}
$$

$$
\begin{aligned}
a \geq 3: \qquad T(n) &= aT(n/2) + bn \\
&= a(aT(n/4) + bn/2) + bn \\
&= a^2 T(n/4) + bn \cdot a/2 + bn \cdot a/2^0 \\
&\quad \vdots \\
&= a^{\log n} T(n/2^{\log n}) + bn \sum_{i=0}^{\log(n)-1} (a/2)^i \\
&= a^{\log n} T(1) + bn \sum_{i=0}^{\log(n)-1} (a/2)^i \\
&= ba^{\log n} + bn \sum_{i=0}^{\log(n)-1} (a/2)^i \\
&= ba^{\log n} + bn \frac{(a/2)^{\log n} - 1}{a/2 - 1} \\
&= bn^{\log a} + bn \frac{\frac{a^{\log n}}{2^{\log n}} - 1}{\frac{a}{2} - 1} \\
&= bn^{\log a} + \frac{b}{\frac{a}{2} - 1} \left(a^{\log n} - n\right) \\
&= bn^{\log a} + \mathcal{O}(a^{\log n})
\end{aligned}
$$

3 Strasse Matrixmultiplikation

nach Aho-Hopfkraft Ulmann

Sei $M_n := \mathbb{R}^{n \times n}$. Wir wollen für $A, B \in M_n$ mit $\mathcal{O}(n^{\log 7})$ arithmetischen Operationen $(+, -, \cdot)$ AB berechnen.

3.1 Prinzip

1. Die $\cdot_2$ Multiplikation von 2×2-Matritzen in beliebigen Ringen ist mit 7 Multiplikationen und 18 Additionen und Subtrakionen realisierbar.[1]

2. Man wendet man diese Verfahren auf R^* an. Nun gilt:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} * \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & \cdots \\ \cdots & \cdots \end{pmatrix} = A * B$$

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} + \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11} + B_{11} & A_{12} + B_{12} \\ A_{21} + B_{21} & A_{22} + B_{22} \end{pmatrix} = A + B$$

Wir skizzieren nun den Beweis für die Multiplikation. Dazu betrachtet man jeden der 4 Quadranten. Für den ersten Quadranten ($i, j \leq n/2$) ergibt sich

$$(A * B)[i, j] = \begin{pmatrix} u & w \end{pmatrix} \begin{pmatrix} v \\ x \end{pmatrix} = (A_{11}B_{11} + A_{12}B_{21})[i, j]$$

wobei u der linke Teil und w der rechte Teil der i-ten Zeile von A und wobei v der obere Teil und x der untere Teil der j-ten Spalte von B ist und $\otimes$ das kanonische Skalarprodukt ist.

Wir wollen nun die Anzahl der arithmetischen Operationen auf $\mathbb{R}$ zur Multiplikation von 4×4 bestimmen. Diese lautet offensichtlich:

$$M(n) = \begin{cases} 1 & n = 1 \\ 7M(n/2) + 18(n/2)^2 & n > 1 \end{cases}$$

Mit einer leichten Anpassung des Master-Theorems, nämlich durch Ersetzung von n durch n^2 im rekursiven Term, erhält man

$$M(n) = \mathcal{O}(n^{\log 7})$$

4 Sortieren

Wir wollen nun **Reihungen (engl. arrays)** nur durch Vergleichen und Umordnung sortieren.

Notation:

- ε ist die leere Reihung
- $\#A$ beschreibt die Länge der Reihung
- $A(i)$ beschreibt das i-te Element der Reihung.
- $A[1 : n]$ beschreibt die Reihung A bzw. eine Auschnitt dieser, die von Stelle 1 bis n geht.

Sei nun

- $\leq$ eine Ordnung auf den Elementen der Reihung A

[1] Der Algorithmus befindet sich auf der Website

- **input:** $A[1:n]$
- **output:** $i < j \to A'(i) \leq A'(j)$
- Zudem sei $S(n)$ die Anzahl der benötigten Vergleiche, um eine Reihung mit einem Verfahren zu sortieren.

Naives Sortieren

$$\begin{aligned}\textsc{Sort}(A) &:= \min A \circ \textsc{Sort}(A \setminus (\min A))\\ \textsc{Sort}(\varepsilon) &:= \varepsilon\end{aligned}$$

Da min für eine Reihung der Länge n, $n-1$ Vergleiche benötigt folgt

$$S(n) = \begin{cases} 0 & \text{für } n = 0 \\ n - 1 + S(n-1) & \text{sonst} \end{cases}$$

sodass $S(n) = \mathcal{O}(n^2)$

4.1 Merge Sort

$$\textsc{Merge}(A[1:n], B[1:m]) := \min(A[1], B[1]) \circ \begin{cases} \textsc{Merge}(A[2:n], B[1:m]) & \text{für } A[1] \leq B[1] \\ \textsc{Merge}(A[1:n], B[2:m]) & \text{für } A[1] > B[1] \end{cases}$$

$$\begin{aligned}\textsc{Merge}(A, \varepsilon) &:= A\\ \textsc{Merge}(\varepsilon, B) &:= B\end{aligned}$$

$$\begin{aligned}\textsc{Sort}(\varepsilon) &:= \varepsilon\\ \textsc{Sort}(A[1:1]) &:= A[1:1]\\ \textsc{Sort}(A[1:k]) &:= \textsc{Merge}(\textsc{Sort}(A[1:k/2]), \textsc{Sort}(A[k/2+1:k]))\end{aligned}$$

Für die Anzahl der Vergleiche von Merge $M(x)$ für $x = n + m$ gilt offensichtlich $M(x) \leq x$. Dementsprechend gilt:

$$\begin{aligned} S(k) &:= \begin{cases} 0 & \text{für } k < 2 \\ 2S(k/2) + M(k/2 + k/2) & \text{sonst} \end{cases} \\ &\leq \begin{cases} 0 & \text{für } k < 2 \\ 2S(k/2) + k & \text{sonst} \end{cases} \end{aligned}$$

Nun folgt offensichtlich aus dem Master-Theorem, dass $S(k) = \mathcal{O}(k \log k)$.

4.2 Quicksort

4.2.1 Exkurs: MfI3

Definition

Wir nennen (S, p) einen **abzählbaren Wahrscheinlichkeitsraum**, wobei S die Menge der möglichen Ergebnisse des Zufallsexperiments beschreibt (sample space) und $p : S \to \mathbb{R}$ die Verteilung beschreibt. Es gilt

$$\sum_{x \in S} p(x) = 1$$

Definition

$A \subseteq S$ heißt **Ereignis**.

Definition

Der **Erwartungswert** E einer Zufallsvariablen X ist wie folgt definiert:

$$E(X) = \sum_{i \in S} X(i) \cdot p(i)$$

Beispiel

Einmaliges Werfen eines Würfels. Es sei

$$S = \{1, \ldots, 6\}$$
$$p(i) = \frac{1}{6}$$
$$A = \{1, 3, 5\}$$
$$B = \{2, 4, 6\}$$

Es gilt also

$$p(A) = \sum_{y \in A} p(y) = \frac{1}{2} = \sum_{y \in B} p(y) = p(B)$$

Der Erwartungswert $E(X)$ mit

$$X : S \to R \qquad\qquad X(y) = y$$

ist

$$E(X) = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + \ldots + 6 \cdot \frac{1}{6} = 3.5$$

Definition

Zwei Ereignisse A, B heißen **unabhängig**, falls

$$p(A \cap B) = p(A) \cdot p(B)$$

Im Beispiel von oben sind die Ereignisse A und B nicht unabhängig, da

$$p(A \cap B) = p(\emptyset) = 0$$
$$p(A) \cdot p(B) = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$$

Es seien nun $W_1 = (S_1, p_1)$ und $W_2 = (S_2, p_2)$ Zufallsexperimente, die unabhängig voneinander durchgeführt werden. Dann ist

$$W_1 \times W_2 = (S_1 \times S_2, q)$$

mit $q(a, b) = p_1(a) \cdot p_2(b)$.

Lemma

$W_1 \times W_2$ ist ein Wahrscheinlichkeitsraum.

Beweis

Es gilt

$$\sum_{(a,b) \in S_1 \times S_2} q(a, b) = \sum_{a \in S_1} \sum_{b \in S_2} p_1(a) \cdot p_2(b) = 1$$

Lemma

Es seinen $A \subseteq S_1$ und $B \subseteq S_2$. Wir wollen zeigen, dass $A \notin S_1 \times S_2$ und $B \notin S_1 \times S_2$ in $W_1 \times W_2$,,unabhängig" sind. (Dieses Lemma ist eigentlich eine Definition.)

Wir definieren e_1 und e_2, sodass $e_1(A)$ und $e_2(B)$ in $W_1 \times W_2$ unabhängig sind (,,Einbettung").

$$e_1(A) := A \times S_2 \qquad e_2(B) := S_1 \times B$$

Beweis

$$\begin{aligned}
q(e_1(A) \cap e_2(B)) &= q((A \times S_2) \cap (S_1 \times B)) \\
&= q(A \times B) \\
&= \sum_{(a,b) \in A \times B} q(a, b) \\
&= \sum_{a \in A} \sum_{b \in B} p_1(a) \cdot p_2(b) \\
&= \sum_{a \in A} p_1(a) \sum_{b \in B} p_2(b) \qquad \text{(mit dem nächsten Lemma)} \\
&= q(e_1(A)) \cdot q(e_2(B))
\end{aligned}$$

Lemma

Es gilt $q(e_1(A)) = p_1(A)$ und $q(e_2(B)) = p_2(B)$.

Es sei $W = (S, p)$ ein Wahrscheinlichkeitsraum und $A_1, \ldots, A_n \subseteq S$ Ereignisse. $A_1, \ldots, A_n$ heißen **gegenseitig unabhängig**, falls

$$\forall K \subseteq \{1, \ldots, n\} : p\Big(\bigcap_{i \in K} A_i \Big) = \prod_{i \in K} p(A_i)$$

Konstruiere $W_i = (S_i, p_i)$, dann ist

$$W_1 \times \ldots \times W_n = (S_1 \times \ldots \times S_n, q)$$

mit $q(a_1, \ldots, a_n) = \prod_{i=1}^{n} p(A_i)$.

Lemma

Es sei $W_1 \times \ldots \times W_n$ ein Wahrscheinlichkeitsraum. Wir können $A_i \subseteq S$ „einbetten“:

$$e_i(A_i) = S_1 \times \ldots \times S_{i-1} \times A_i \times S_{i+1} \times \ldots \times S_n$$

Lemma

Es sei $A_i \subseteq S_i$ mit $i = 1, \ldots, n$. Dann sind $e_i(A_i)$ gegenseitig unabhängig.

4.2.2 Algorithmus und Komplexität

Es sei

$$T(n) = E(\#\text{Vergleiche von } \textsc{Qsort}(A) \text{ mit } |A| = n)$$
$$W_n = \text{Wahrscheinlichkeitsraum für } \textsc{Qsort}(A) \text{ mit } |A| = n$$

$$\begin{array}{rl} & \textsc{Qsort}(A[1:n]) \\ & y = \textsc{Random}(n) \\ & \text{Splitter: } A[y] \qquad\qquad \text{(Pivot)} \\ A_< = & (A[i] : A[i] < A[y]) \\ A_= = & (A[i] : A[i] = A[y]) \\ A_> = & (A[i] : A[i] > A[y]) \\ & \text{Output: } \textsc{Qsort}(A_<) \circ A_= \circ \textsc{Qsort}(A_>) \end{array}$$

wobei $A_<$, $A_=$ und $A_>$ je $n - 1 \leq n$ Vergleiche benötigen.

Lemma

Es ist

$$T(n) \leq \underbrace{n}_{\text{Splitter}} + \frac{1}{n} \sum_{i=1}^{n} (T(i) + T(n-i))$$

mit $i = |A_<|$.

4.2.3 Exkurs: MfI3 -Teil 2

Definition

Es seien $X_1 : S_1 \to \mathbb{R}$ und $X_2 : S_2 \to \mathbb{R}$ Zufallsvariablen für W_1 beziehungsweise W_2. Und seien $\tilde{X_1} : S_1 \times S_2 \to \mathbb{R}$ sowie $\tilde{X_2} : S_1 \times S_2 \to \mathbb{R}$ ebenfalls Zufallsvariablen. Es gilt

$$\tilde{X_1}(a, b) = X_1(a)$$

$$\tilde{X_2}(a,b) = X_2(b)$$

Lemma

$$E(\tilde{X_1}) = E(X_1)$$
$$E(\tilde{X_2}) = E(X_2)$$

Beweis

E + D (Expandiere Definition und Distributivgesetz)

Lemma

$$E(\tilde{X_1} + \tilde{X_2} + c) = c + E(X_1) + E(X_2)$$

Definition

Sei $W = (S, p)$ ein Wahrscheinlichkeitsraum. Für $A, B \subseteq W$ nennt man

$$p(A|B) = \frac{p(A \cap B)}{p(B)}$$

bedingte Wahrscheinlichkeit von A gegeben B.

Beispiel

Werfen eines Würfel mit den Ereignissen $B = \{2, 4, 6\}$, $A = \{1\}$ und $C = \{2\}$.

$$p(A|B) = \frac{0}{\frac{1}{2}} = 0$$

$$p(C|B) = \frac{\frac{1}{6}}{\frac{1}{2}} = \frac{1}{3}$$

4.2.4 2-Phasen-Experiment

Definition

Experiment 1: Sei $W = (S, p)$ ein Wahrscheinlichkeitsraum und $i \in W$. $i \mapsto w_i(S_i, p_i)$
Das Experiment w_i wird ausgeführt, falls das Ergebnis des ersten Experiments = i. Vorausgesetzt $S_i \cap S_j = \emptyset$ und $i \neq j$.

$$Q = w \to \{w_i\}$$
$$Q = (\bigcup_{i \in S} \{i\} \times S_i, q)$$
$$q(i \in \{i\}, a \in S_i) = p(i) \cdot p_i(a)$$

$$\tilde{X} : \bigcup_{i \in S} \{i\} \times S_i \to \mathbb{R}, (i, a) \mapsto X_i(a)$$

wobei $X_i : S_i \to \mathbb{R}$.

Lemma

Q ist ein Wahrscheinlichkeitsraum.

Beweis

E + D

Lemma

Zusammenhang mit bedingter Wahrscheinlichkeit

$$q(\{i\} \times A \subseteq S_i | \{i\} \times S_i) = p_i(A)$$

Lemma

$$E(\tilde{X}) = \sum_{i \in S} p(i) \cdot E(X_i)$$

4.2.5 Algorithmus und Komplexität 2

$$\begin{aligned} \textsc{Qsort}(\varepsilon) &= \varepsilon \\ \textsc{Qsort}(A[1]) &= A[1] \\ \textsc{Qsort}(A[1:2]) &= \begin{cases} A[1:2] \text{für} A[1] \leq A[2] \\ A[2] \circ A[1] \end{cases} \end{aligned}$$

$$\begin{gathered} \textsc{Qsort}(A) |A| \geq 3 \\ y = \textsc{Random}(n) \\ \text{splitter} = A[y] \quad \text{(Pivotelement)} \\ A_< = (A[k] | A[k] < A[y]) \\ A_= = (A[k] | A[k] = A[y]) \\ A_> = (A[k] | A[k] > A[y]) \\ \text{Output: } \textsc{Qsort}(A_<) \circ A_= \circ \textsc{Qsort}(A_>) \end{gathered}$$

4.2.6 Q_n Raum für $\textsc{Qsort}(A)$ mit $|A| = n$

Sei $Q_n = w \to \{w_i\}$. Man nehme an A[i] sei paarweise verschieden.

1. Experiment:
$w = ([1:n], p)$
$i = |A_<| + 1$(Rang vom Splitter in A)

Folgeexperimente:

i	w_i
1	Q_{n-1}
2	Q_{n-2}
$3 \leq i \leq n-2$	$Q_{i-1} \times Q_{n-i}$
n - 1	Q_{n-2}
n	Q_{n-1}

$$T(n) = E(\#\text{Vergleiche für } |A| = n)$$

$$T(n)\ E(X)\ X : \bigcup_i \{i\} \times \underbrace{S_i}_{Ereignisraum} \quad \rightarrow N\#Vergleiche$$

mit Lemma 5 und 8 folgt

$$T(n) = n - 1 + \frac{2}{n} \cdot T(n-1) + \frac{2}{n} \cdot T(n-2) + \frac{1}{n} \sum_{i=3}^{n-2} (T(i-1) + T(n-i)$$

$$T(n) \leq n + \frac{2}{n} \sum_{i=2}^{n-1} T(i)$$

Satz

$T(n) = O(nlogn)$

Beweis

Induktionsbehauptung

$$T(n) \leq 2n \cdot ln(n)$$

$$ln(2) = 0.69 \leq 0.7(\text{nötig für Abschätzungen})$$

Induktionsanfang

$$n = 2: \ T(2) = 1$$

$$2 \cdot 2ln(2) \leq 4 \cdot 0.7 = 2.8$$

Induktionsschritt

$$n \geq 3 \quad n-1 \to n$$

$$T(n) \leq n + \frac{2}{n} \sum_{i=2}^{n-1} T(i)$$

$$\leq n + \frac{4}{n} \sum_{i=2}^{n-1} i \cdot ln(i) \qquad \text{(IV)}$$

$$\leq n + \frac{4}{n} \int_2^n x \cdot ln(x) dx$$

$$= n + \frac{4}{n} \underbrace{[\underbrace{\frac{x^2}{2} ln(x) - \frac{x^2}{4}}_{\frac{n^2}{2} ln(n) - \frac{n^2}{4} - (\frac{4}{2} ln(2) - 1)}]_2^n}_{\leq \frac{n^2}{2} \cdot ln(n) - \frac{n^2}{4}}$$

$$T(n) \leq n + \frac{4}{n}(\frac{n^2}{2} ln(n) - \frac{n^2}{4}) = 2nln(n)$$

4.3 Alternativ-Beweis zu Quicksort

Definiere

$$\begin{aligned} S_0 &= \{\star\} \\ S_n &= \bigcup_{i=0}^{n-1} \{i\} \times S_i \times S_{n-i-1} \end{aligned}$$

$$\begin{aligned} p_0(\star) &= 1 \\ p_n(i,x,y) &= \frac{1}{n} p_i(x) p_{n-i-1}(y) \end{aligned}$$

$$\begin{aligned} X_0(\star) &= 0 \\ X_n(i,x,y) &= n - 1 + X_i(x) + X_{n-i-1}(y) \end{aligned}$$

$$T(n) = \mathbb{E}\left[X_n\right]$$

Wir führen Induktion mit der Hypothese

$$\forall n \in \mathbb{N} : T(n) \leq \begin{cases} 0 & n = 0 \\ 2n \log n & n \neq 0 \end{cases}$$

Induktionsbasis

$$\begin{aligned} T(0) &= \mathbb{E}\left[X_0\right] \\ &= X_0(\star) p_0(\star) \\ &= 0 \end{aligned}$$

Induktionsschritt

$$\begin{aligned} T(n) &= \mathbb{E}\left[X_n\right] \\ &= \sum_{(i,x,y) \in S_n} X_n(i,x,y) p_n(i,x,y) \\ &= \sum_{(i,x,y) \in S_n} \Big(n - 1 + X_i(x) + X_{n-i-1}(y)\Big) \frac{1}{n} p_i(x) p_{n-i-1}(y) \end{aligned}$$

Betrachte die Teilterme in der großen Klammer.

$$\sum_{(i,x,y) \in S_n} n \frac{1}{n} p_i(x) p_{n-i-1}(y) = n \underbrace{\sum_{(i,x,y) \in S_n} \frac{1}{n} p_i(x) p_{n-i-1}(y)}_{=1} = n$$

$$\sum_{(i,x,y) \in S_n} \frac{1}{n} p_i(x) p_{n-i-1}(y) = \sum_{(i,x,y) \in S_n} p_n(i,x,y) = 1$$

$$\begin{aligned}
\sum_{(i,x,y)\in S_n} X_i(x)\frac{1}{n}p_i(x)p_{n-i-1}(y) &= \sum_{i=0}^{n-1}\sum_{x\in S_i}\sum_{y\in S_{n-i-1}} X_i(x)\frac{1}{n}p_i(x)p_{n-i-1}(y) \\
&= \frac{1}{n}\sum_{i=0}^{n-1}\underbrace{\sum_{x\in S_i} X_i(x)p_i(x)}_{=\mathbb{E}[X_i]}\underbrace{\sum_{y\in S_{n-i-1}} p_{n-i-1}(y)}_{=1} \\
&= \frac{1}{n}\sum_{i=0}^{n-1}\mathbb{E}\left[X_i\right] \\
&= \frac{1}{n}\sum_{i=0}^{n-1}T(i)
\end{aligned}$$

$$\begin{aligned}
\sum_{(i,x,y)\in S_n} X_{n-i-1}(y)\frac{1}{n}p_i(x)p_{n-i-1}(y) &= \sum_{i=0}^{n-1}\sum_{x\in S_i}\sum_{y\in S_{n-i-1}} X_{n-i-1}(y)\frac{1}{n}p_i(x)p_{n-i-1}(y) \\
&= \frac{1}{n}\sum_{i=0}^{n-1}\underbrace{\sum_{y\in S_{n-i-1}} X_{n-i-1}(y)p_{n-i-1}(y)}_{=\mathbb{E}[X_{n-i-1}]}\underbrace{\sum_{x\in S_i} p_i(x)}_{=1} \\
&= \frac{1}{n}\sum_{i=0}^{n-1}\mathbb{E}\left[X_{n-i-1}\right] \\
&= \frac{1}{n}\sum_{i=0}^{n-1}T(n-i-1)
\end{aligned}$$

$$\begin{aligned}
\implies T(n) &= n-1+\frac{1}{n}\sum_{i=0}^{n-1}T(i)+\frac{1}{n}\sum_{i=0}^{n-1}T(n-i-1) \\
&= n-1+\frac{1}{n}\sum_{i=0}^{n-1}T(i)+\frac{1}{n}\sum_{i=0}^{n-1}T(n-(n-1-i)-1) \\
&= n-1+\frac{2}{n}\sum_{i=0}^{n-1}T(i) \\
&= n-1+\frac{2}{n}\sum_{i=2}^{n-1}2i\log i \qquad \text{// } T(0)=T(1)=0 \text{ nach Induktion} \\
&\leq n-1+\frac{2}{n}\int_2^n 2x\log x\ \mathrm{d}x \\
&= n-1+\frac{1}{n}\left[x^2(2\log(x)-1)\right]_2^n \\
&= n-1+\frac{1}{n}\left[n^2(2\log(n)-1)-\underbrace{2^2(2\log(2)-1)}_{=4}\right] \\
&= n-1+2n\log(n)-n-\frac{4}{n} \\
&\leq 2n\log(n)
\end{aligned}$$

■

4.3.1 Anmerkung

Sind nlog(n) Vergleiche zum Sortieren notwendig?

Antwort:

Im Allgemeinen gilt für A[1:n]: $A[i] \neq A[j]$ für $i \neq j$
Input: $\pi(\text{SORT}(A)), \pi \in \{\text{Bijektionen } \{1 \dots \#A\} \to \{1 \dots \#A\}\} = S_{\#A}$
Output: $\pi^{-1}(\text{in}) = \pi^{-1}(\pi(\text{SORT}(A))) = \text{SORT}(A)$

$$
\begin{aligned}
\#\text{Blätter} &\geq \#\{\pi^{-1} \mid \pi^{-1} \in S_{\#A}\} \\
&= \#\{\pi \mid \pi \in S_{\#A}\} \\
&= n! \\
&= \underbrace{n(n-1)(n-2)...(\frac{n}{2})}_{\geq (\frac{n}{2})^{\frac{n}{2}}}(\frac{n}{2}-1)...1 \\
&\Rightarrow 2^T \geq (\frac{n}{2})^{\frac{n}{2}} \\
&\Rightarrow T \geq \frac{n}{2}\log(\frac{n}{2})
\end{aligned}
$$

4.4 Zusammenfassung der Komplexitäten

1) int mul: naiv in $n^2 \Rightarrow n^{1,59}$ mit Karatsuba
2) Matrix mul: naiv in $n^3 \Rightarrow n^{2,81}$ mit Strasser-Mult.
3) sort in $n^2 \Rightarrow n \log n$
4) quicksort mit W-Raum: Erwartungswert: $n \log n$
5) Für sort gilt allgemein: $\geq n \log n$

5 Median mit O(n) Vergleichen (nach Blum, Tarjan, u.a.)

5.1 Einführung

Betrachte Liste A[1:n] mit $x \in A$

Definition

Der Rang eines Elementes x einer Liste A ist definiert als

$$rang(x, A) = \#\underbrace{\{i|A[i] < x\}}_{A_<} + 1$$

Definition

Sei $i \in [1 : n]$. Select von i und A bedeutet:

$$select(i, A) = x \Leftrightarrow rang(x, A) = i$$

wobei $x \in A$.

Definition

Der Median einer Liste A der Länge n med(A) ist definiert als:

$$sort(A)[\lfloor \frac{n}{2} \rfloor + 1]$$

Lemma

$$med(A) = select(\lfloor \frac{n-1}{2} \rfloor + 1, A)$$

5.2 Rekursiver select-Algorithmus

Folgender Algorithmus berechnet select(i,A) rekursiv:

1. $n < 60$: erst sortieren, dann entsprechendes Element wählen
2. Bilde $r = \lfloor \frac{n}{5} \rfloor$ 5er Gruppen aus Elementen der Liste. Hierbei können bis zu 4 Elemente übrig bleiben.
3. Sortiere die 5er Gruppen (und die Restgruppe) durch Insertion Sort und bestimme ihren Median.
4. Rekursion: Bestimme m = Median der Mediane der 5er Gruppen
5. split A durch m: Liste wird zu $A_< \circ m \circ A_>$
6. Für $|A_<| = i - 1$: out: m
   Für $|A_<| > i - 1$: out: $select(i, A_<)$
   Für $|A_<| < i - 1$: out: $select(y, A_>)$, wobei $y = i - |A_<| - 1$

Lemma

Sei $q \in \{1, ..., n\}$ mit $A_< \circ m \circ A_>[q] = m$.
Dann gilt: $q - 1 \geq \frac{3n}{10} - 3$ und $n - q \geq \frac{3n}{10} - 3$
Desweiteren gilt: $max\{|A_<|, |A_>|\} \geq \lceil \frac{7n}{10} \rceil + 2$
Beweis

Es gilt:
$q - 1 = |A_<| \geq \#U = \lfloor \frac{r-1}{2} \rfloor \cdot 3 + 2$

$n - q = |A_>| \geq \#V = \lfloor \frac{r-1}{2} \rfloor \cdot 3 + 2$

$$\begin{aligned}
\lfloor \frac{r-1}{2} \rfloor \cdot 3 + 2 &\geq 3\frac{r-2}{2} + 2 \\
&\geq \frac{3}{2}(\frac{n}{5} - \frac{4}{5} - 2) + 2 \\
&= \frac{3n}{10} - \frac{3 \cdot 4}{2 \cdot 5} - 3 + 2 \\
&= \frac{3n}{10} - 2.2 \\
&\geq \frac{3n}{10} - 3
\end{aligned}$$

Damit gilt für die zweite Aussage:
$max\{|A_<|, |A_>|\} \geq \lceil n - (\frac{3n}{10} - 3) - 1 \rceil \geq \lceil \frac{7n}{10} \rceil + 2$

Mit obigem Lemma gilt folgendes:
Für t(n)=maximale Vergleiche für select (i,A) mit dem beschriebenen Algorithmus, wobei $n = |A|$ existieren Konstanten d und e, sodass:

$$t(n) \leq \begin{cases} t(\lfloor \frac{n}{5} \rfloor) + t(\lceil \frac{7n}{10} \rceil + 2) + d \cdot n & \text{für } n \geq 60 \\ e \cdot n & \text{sonst} \end{cases}$$

Bemerkung

Die Formel für t(n) resultiert aus folgenden Beobachtungen:

1. Für $n < 60$ gilt mit merge sort, es existiert ein a, sodass
   $t(n) \leq a \cdot (logn) \cdot n \leq \underbrace{a \cdot (log60)}_{:=e} \cdot n$
2. $t(\lfloor \frac{n}{5} \rfloor)$ ist die Zahl an Vergleichen in der Rekursion in Schritt 3.
3. $t(\lceil \frac{7n}{10} \rceil + 2)$ ist die Zahl an Vergleichen in der Rekursion in Schritt 5.
4. $d \cdot n$ sind die nötigen Vergleiche für die Sortierungen der 5er-Gruppen.

Lemma

Es gilt: $\exists c \in \mathbb{N} : t(n) \leq c \cdot n$ **Beweis**

Beweis durch Induktion über $n \in \mathbb{N}$:
IA: Für $n < 60$ folgt dies sofort aus der Definition.
IS: Es gilt folgende Abschätzung:

$$\begin{aligned}
t(n) &\leq c\lfloor \frac{n}{5} \rfloor + c(\lceil \frac{7n}{10} + 2 \rceil + 2) + dn \\
&\leq c\frac{n}{5} + c\frac{7n}{10} + 3c + dn \\
&\leq nc\frac{9}{10} + 3c + dn
\end{aligned}$$

Mit $c \geq 20d \Leftrightarrow d \leq \frac{c}{20}$ und $n > 60$ gilt dann:

$$\begin{aligned} 3c + dn &\leq 3c + \frac{c}{20}n \\ &= c(3 + \frac{n}{20}) \\ &\leq c(3 \cdot \frac{n}{60} + \frac{n}{20}) \\ &= c\frac{n}{10} \end{aligned}$$

und damit

$$t(n) \leq nc\frac{9}{10} + c\frac{n}{10} = cn.$$

Somit ist die Laufzeit des Median-Algorithmus in $\mathcal{O}(n)$.

Datenstrukturen

Im folgenden werden wir programmiersprachliche Konzepte benötigen:

1. records
2. pointer
3. Zuweisungen v = e, wobei e beliebige arithmetische Ausdrücke sein können
4. if-Statements
5. for- und while-Schleifen
6. Funktionsaufrufe

Die Laufzeit entspricht nun also der Zeit des compilierten Programms auf einer idealisierten MIPS-Maschine.

5.3 Rekursive Datentypen

Rekursive Datentypen sind nicht durch eine normale Rekursion definiert.

1. normal:
$$R(i) = F(R(i-1)...) = F(R(i-1), R(i-2), ..., R(i))$$
2. was man nicht haben möchte:
$$F(..., R(i+1), ...)$$

5.4 Variablendeklaration

1. $t\ x$:
   (a) t: Typ
   (b) x: Name der deklarierten Variable
2. $t \in E$:
   $$E = \{int, float, char, bool, ...\} \qquad \text{„elementare Datentypen"}$$
3. $E \cup T$:
   T ist eine Menge von Typen durch Folge $td_1, ..., td_i, ..., td_N$ von Typendefinitionen deklariert
4. td_i:
   $$\text{typedef} \quad \underbrace{d}_{\text{Typ-Deskription (arrays, structs, pointer)}} \quad \underbrace{t}_{\text{Name des deklarierten typs}}$$

- arrays: $t'(n)\ t \qquad$ wobei $t' \in T_{i-1}$
- structs: $\{t_1 n_1, ..., t_s n_s\} \qquad t_j \in T_{i-1}$ und $1 \leq j \leq s$
- pointer $t'*\ \ t$
  1. pointer dereferenzieren: $x*$ liefert in Ausdrücken die Variable, die auf den Pointer zeigt
  2. in C: malloc
  3. in PASCAL:
     $$x = \text{new } t'*$$
     dabei wird eine namenlose Variable vom Typ t' erzeugt und x weist pointer auf t' zu.
  4. Bei typedef $t' * t$ muss $t' \in T$ erlaubt werden.

5.5 Syntaktischer Zucker

1. struct LEL (Listenelemente):
   $$\{.., cont, \text{LEL} *\ next\}\ \text{LEL}$$
2. struct TEL (Treeelemente):
   $$\text{TEL} * u$$
   $$\{.., cont, u\ lson, u\ rson, u\ parent\}$$

3. Frage: Warum geht das insbesondere beim Compilieren gut?
   $\Rightarrow$ Compiler plant Platz für Variablen

$$t \mapsto \text{size}(t)$$
$$\text{64-Bit-Maschine}$$
$$t'(n)\ t$$
$$\{t_1 n_1, ..., t_s n_s\}\ t$$

$$\#\text{bytes für Variable vom Typ } t$$
$$t \in E : \text{size}(t) = 8 \qquad t' *\ t$$
$$\text{size}(t) = n \cdot \text{size}(t') \qquad \text{unabhängig von Deklaration von } t'$$
$$\text{size}(t) = \sum_{i=1}^{s} \text{size}(t_1)$$
$$\text{size}(t) = 8$$

6 Heap-Sort

6.1 Heap

Ein Heap ist ein fast vollständiger binärer Baum, bei dem alle Level bis auf das unterste komplett aufgefüllt sind. Dabei stellt jeder Knoten ein Element in einem array dar. Wir nummerieren von oben nach unten und von links nach rechts.

6.2 Implementierung von Heap-Sort

1. struct TEL (Treeelemente) mit den Komponenten: cont, lson, rson, parent
2. Sei A ein array der Länge n: $A[1:n]$
   - $n = \#$ Knoten im Heap, wobei gilt: $2^{k-1} < n \leq N = 2^k$ als kleinste 2-er Potenz.
   - heapsize $n = 2^k$
   - Wir haben 3 Funktionen:
     (a) $lson(i) = 2i$
     (b) $rson(i) = 2i + 1$
     (c) $parent(i) = \lfloor \frac{i}{2} \rfloor$
   - heap property:
     Es gilt:
     (a) $A(i) \geq A(lson(i))$
     (b) $A(i) \geq A(rson(i))$
     (c) $A(i) \leq A(parent(i))$
3. Unsere Implementierung besteht aus 3 Teilen:

(a) heapify (um heap property zu verbessern):

- Input: Heap A, und Index i, sodass die heap property erfüllt ist für alle Knoten in den Subheaps mit den Wurzeln Left$[i] = lson(i)$ und Right$[i] = rson(i)$.
- Output: Am Ende ist die heap property für i erfüllt.

```
1  l := Left[i]
2  r := Right[i]
3  h := i
4  if l ≤ heapsize and A[l] > A[i] then
5          h = l
6  if r ≤ heapsize and A[r] > A[h] then
7          h = r
8  if h ≠ i then
9          swap(A[i], A[h])
10         Heapify(A,h)
```

- Laufzeit: $\mathcal{O}(\text{Höhe}(i))$

(b) Makeheap (heap property global herstellen):

i. Input: $A[1:n]$

ii. Output: A erfüllt die heap property

```
1  heapsize := n
2  for i = ⌊n/2⌋, ..., 1 do
3  Heapify(A, i)
```

iii. Invariante: heap property(j) für alle $j \geq i$

iv. Laufzeitanalyse:

v. Laufzeit:

$$\begin{aligned} t(n) &\leq \mathcal{O}(\frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + ...) \\ &\leq n \cdot \mathcal{O}(\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + ...) \\ &\leq \sum_{i=1}^{\infty} \frac{i}{2^i} \\ &\leq \sum_{i=1}^{i_0-1} \frac{i}{2^i} + \sum_{i=0}^{\infty} (\frac{2}{3})^i \\ &= \mathcal{O}(i) \end{aligned}$$

$$i \leq (\frac{4}{3})^i \Leftrightarrow \frac{i}{2^i} \leq (\frac{2}{3})^i \qquad \forall i \geq i_0$$

$$\longrightarrow t(n) \leq \mathcal{O}(n)$$

(c) Heap sort:

i. Input: $A[1:n]$

ii. Output: A ist sortiert

iii. Invariante: $A[i:n]$ ist sortiert und enthält die $n-i+1$ größten Elemente

```
1 heapsize := n
2 Makeheap(A)
3 for i := n, n−1,..., 2 do
4     Swap(A[1], A[i])
5     heapsize := heapsize − 1
6     Heapify(A, 1)
```

iv. Laufzeit: $\mathcal{O}(n \cdot log(n))$

7 Binary Search Trees (BST)

u = p(x) - "Parent von x"
z = l(x) - "Left-Son von x"
y = r(x) - "Right-Son von x"

Definition

TEL = Tree-Element

Implementierung:
Realisierung durch Structs:
struct $\{\text{int key}, \text{TEL } x, l, r, p\}$,
wobei TEL x ein Pointer auf ein Tree-Element darstellt

Variablen vom Typ TEL leben auf dem Laufzeitheap und haben keine Namen.
Implementierung: Pointervariable TEL* x, bezeichnet einen Pointer auf den Knoten x

$$p(x) = (x*)p \qquad l(x) = (x*)l \qquad r(x) = (x*)r$$

Definition

Wir definieren einige sinnvolle Prädikate:

(i) T(x) = Baum mit Wurzel x

(ii) isr(x) = x isRightSon $\equiv$ (r(p(x)) = x)

(iii) isl(x) = x isLeftSon

(iv) isRoot(x) $\equiv$ p(x) $=\perp$, wobei $\perp$ ein NULL-Pointer ist

(v) isLeaf(x) $\equiv$ l(x) = r(x) = $\perp$

(vi) h(T) = h(w) = Höhe von w in T (= Distanz zwischen Wurzel und unterstem Knoten)

(vii) T(x) = Teilbaum mit Wurzel x

(viii) L(x) = T(l(x)) bezeichnet den linken Unterbaum der als Wurzel den linken Sohn von x hat

(ix) R(x) = T(r(x)) analog

(x) x $\in$ T $\equiv$ x ist Knoten in Baum T

7.1 Binary Search Trees-BST

Definition

Seien $x, y \in$ T

(i) Wenn x $\neq$ y $\rightarrow$ key (x) $\neq$ key(y)

(ii) y $\in$ L(x) $\rightarrow$ key (y) $<$ key (x)

(iii) y $\in$ R(x) $\rightarrow$ key (x) $<$ key (y)

Ein Baum ist genau dann ein BST wenn alle drei Eigenschaften erfüllt sind.

Bemerkung

BST(T) $\equiv \forall$ x $\in$ T : BST(x)
(wobei x ein Knoten in T, also muss jeder einzelne Knoten die BST Eigenschaften erfüllen)

7.1.1 Parent Chasing

Definition

1. $p^0(x) = x$
2. $p^{i+1} = p(p^i(x))$ (wobei p^{n-1} die Wurzel ist, wenn der Baum die Höhe n hat)

Definition

Seien x,y Wurzeln zweier Unterbäume eines kleinsten BST mit Wurzel a(x,y), wobei x $\notin$ T(y) und y $\notin$ T(x). Dann ist a(x,y) der tiefste gemeinsame Vorfahre von x und y.

Lemma 1

Sei y $\in$ T(x). Dann gilt key(y) $<$ key(x) $\iff$ y $\in$ L(x)
Beweis

Folgt aus den BST Eigenschaften.

Lemma 2

Sei y $\notin$ T(z) und z $\notin$ T(y) Dann gilt key(y) $<$ key(z) $\iff$ y $\in$ L(a(z,y)) $\wedge$ z $\in$ R(a(z,y))

7.1.2 Operationen auf BST's

1. **Search**
   Search(x,k) liefert zu x $\in$ T und einem key k den entsprechenden Knoten
   Input : x $\in$ T, k $\in \mathbb{Z}$: key

   $$\text{Output}=\begin{cases} y \in T \text{ mit key}(y) = k & \text{falls der Knoten existiert} \\ \perp & \text{sonst} \end{cases}$$

```
1 if key(x) = k or x = ⊥
2         return x
3 else {
4         if k < key(x)
5                 return search(l(x),k)
6         else
7                 return search(r(x),k)
8 }
```

   Laufzeit: $\mathcal{O}$(h(x))

   Beweis Korrektheit von Search

   Beweis durch Induktion über h(x):
   Basisfall: h(x) = 0 $\rightarrow$ Blatt ✓
   Induktionsannahme: die Korrektheit gilt bereits für alle induktiv kleineren Teilbäume
   h $\rightarrow$ h+1
   k $<$ key(x) $\rightarrow$ y, falls er existiert in L(x)
   k $>$ key(x) $\rightarrow$ y, falls er existiert in R(x)
   IA $\rightarrow$ ✓

2. **Minimum/Minimierung? min(x)**
   Input: x $\in$ T
   Output: y $\in$ T(x) mit key(y) = min {key(z) : z $\in$ T(x)}

```
1 if l(x) = ⊥ then
2     return x
3 else
4     return min(l(x))
```

   Beweis

   Korrektheit folgt aus den BST Eigenschaften
   Laufzeit: $\mathcal{O}$(h(x))

3. **Successor**
succ(x)
Input: x ∈ T
Output=$\begin{cases} \text{y} \in \text{T key(x)} < \text{key(y)} \wedge \nexists \text{z} \in \text{T: key(x)} < \text{key(z)} < \text{key(y)} & \text{falls y existiert} \\ \bot & \text{sonst} \end{cases}$
Sei im Folgenden i = min $\{j : isl(p^j(x))\}$

```
1 if r(x) ≠ ⊥
2  return min(r(x))
3 if r(x) = ⊥
4  return p^{i+1}(x)
```

Beweis Korrektheit

FU:

(a) r(x) ≠ ⊥
BST: y ∈ R(x) → key(x) < key(y)
Beweis durch Widerspruch :
Annahme: ∃ z ∈ T : key(x) < key(z) < key(y)

i. z ∈ R(x)
z ≠ y
→key(y) < key(z) ↯

ii. x ∈ T(z)
Dann ist per Annahme:
key(x) < key(z)
→ x ∈ L(z) Lemma 1
→ y ∈ L(z)
→ key(y) < key(z) BST-Eigenschaft
↯

iii. x ∉ T(z) ∧ z ∉ T(x)
key(x) < key(z)
→ z ∈ R(a(x,z)) ∧ x ∈ L(a(x,z)) Lemma 2
→ y ∈ L(a(x,z))
→ key(y)< key(z) ↯

(b) r(x) = ⊥
BST : x ∈ L(y) → key(x)<key(y)
Annahme: ∃ z : key(x)<key(z) <key(y)

i. z ∈ T(p^j(x)) \ T(p^{j-1}(x))
Dann ist z ∈ L(p^j(x)) ∧ x ∈ R(p^j(x)) → key(z) < key(x)↯

ii. x ∈ T(z)
x ∉ T(z) ∧ z ∈ T(x)
analog

7.1.3 Einfügen und Weglassen

(a) **(Naives) Einfügen**
insert(x,z) fügt den Knoten z falls noch nicht vorhanden in Baum x ein
Input: x∈ T, z = Knoten
Output: T' mit(eventuell) extra Knoten z,d.h. ∃ y ∈ T: key(y) = key(z) → T' = T

```
1  if x = ⊥ then root = z
2    else
3  if key(z)< key(x) then{
4    if l(x) = ⊥ then {l(x) = z
5                      p(z) = x}
6    if l(x) ≠ ⊥ then insert (l(x),z)}
7  if key(x) < key(z) then{
8    if r(x) = ⊥ then {r(x) = z
9                      p(z) = x}
10   else insert(r(x),z)}
```

Laufzeit: O(h(x))
Problem: worst Case ist eine sortierte Folge ergibt n Knoten und Tiefe n-1

(b) **Delete(x)**
x ∈ T
Output: T', (y ∈ T' $\iff$ y ∈ T ∧ y ≠ x) ∧ BST (T')
3 Fälle :

i. isLeaf(x) (x hat keinen Sohn)

```
1  if isRoot(x) then root =⊥
2  if isl(x) then l(p(x))= ⊥
3  if isr(x) then r(p(x))= ⊥
```

ii. x hat einen Sohn

```
1  if ¬ (isRoot(x)) then {
2   if r(x) ≠ ⊥∧ isr(x) {
3       r(p(x))= r(x)
4       p(r(x))= p(x)}
5  }
6  else {
7    if r(x) ≠ ⊥ then {
8       p(r(x)) = ⊥
9       root = r(x)
10   }
11   else if l(x) ≠ ⊥ then {
12      p(l(x)) = ⊥
13      root = l(x)
14   }
15 }
```

(für left analog)

iii. r(x)$\neq \perp \wedge$ l(x) $\neq \perp$

```
1 y = succ(x)
2 if isl(y) then l(p(y))= ⊥
3 if isr(y) then r(p(y))= ⊥
4 l(y) = l(x)
5 r(y) = r(x)
6 p(y) = p(x)
7 if isr(x) then r(p(x)) = y
8 if isl(x) then l(p(x)) = y
```

7.2 Bemerkung

1. $L(min(x)) = \perp$
2. $succ(x)$: sollte $r(x) \neq \perp \implies succ(x) = min(r(x)) \implies l(succ(x)) = \perp$ (wegen minimum!)

8 AVL-Bäume

Ziel: Wir wollen Bäume mit n Knoten und einer Tiefe von $\mathcal{O}(logn)$ bei denen *search*, *insert*, *delete*, und *bal* (balanciertheit) in $\mathcal{O}(logn)$.
Hierfür definieren wir uns *bal* wie folgt:
$bal(x) \equiv h(l(x)) - h(r(x))$

8.1 AVL-Eigenschaft

Ein Binärbaum erfüllt die AVL-Eigenschaft genau dann, wenn

1. er die Binary-Search-Tree-Eigenschaft erfüllt
2. für jeden Knoten x gilt $bal(x) \in \{-1, 0, 1\}$

8.2 Ghost-nodes

Allerdings brauchen wir für die Definiertheit von $bal(x)$ für jedes x einen Sohn. Aus diesem Grund definieren wir uns zwei Arten von Knoten:

1. implementierungs (reale Knoten) $\circledcirc$
2. ghosts (im Code NUR für Analyse) $\square$

Dabei ist wichtig, dass es keinen Informationsfluss von den ghosts zu den realen Knoten gibt. Somit hat jetzt jeder Implementierungsknoten zwei Söhne.

8.3 Lemma 1:

Sei T ein AVL-Baum, n die Anzahl an Implementierungsknoten und h die Höhe. Dann gilt: $h \in \mathcal{O}(logn)$
Für den Beweis brauchen wir noch 2 Hilflemmata.

8.4 Lemma 2:

Die n-te Fibonacci-Zahl lässt sich auch auf die folgende Form bringen:
$F_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n$

$P(n)$ sei die Aussageform

$$f(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n \right)$$

Induktionsanfang Beweis von $P(0)$
Es ist $f(0) = 0 = \frac{1}{\sqrt{5}}(1-1)$ sodass $P(0)$ richtig ist.

Beweis von $P(1)$
Es ist $f(1) = 1 = \frac{1}{\sqrt{5}}(\sqrt{5}) = \frac{1}{\sqrt{5}}(\frac{2\sqrt{5}}{2})$ sodass $P(1)$ richtig ist.

Induktionsschluss
Wir beweisen, dass die Induktionsvoraussetzung

$$\forall m < n : P(m)$$

die Induktionsbehauptung $P(n)$ impliziert.

Es gilt:

$$\begin{aligned} f(n) &= f(n-1) + f(n-2) \\ &= \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1} \right) + \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2} \right) \end{aligned}$$

nach Induktionsvoraussetzung.

Wir substituieren:

$$a = \left(\frac{1+\sqrt{5}}{2}\right)$$
$$b = \left(\frac{1-\sqrt{5}}{2}\right)$$

Damit gilt:

$$
\begin{aligned}
f(n) &= \frac{1}{\sqrt{5}}\left(a^{n-1} - b^{n-1}\right) + \frac{1}{\sqrt{5}}\left(a^{n-2} - b^{n-2}\right) && \text{Distributivität} \\
&= \frac{1}{\sqrt{5}}(a^{n-1} - b^{n-1} + a^{n-2} - b^{n-2}) && \text{Komm., Ass.} \\
&= \frac{1}{\sqrt{5}}((a^{n-1} + a^{n-2}) - (b^{n-1} + b^{n-2})) && \text{Potenzgesetze} \\
&= \frac{1}{\sqrt{5}}(a^n(a^{-1} + a^{-2}) - b^n(b^{-1} + b^{-2}))
\end{aligned}
$$

Es gilt weiterhin:

$$
\begin{aligned}
a^{-1} + a^{-2} &= \left(\frac{1+\sqrt{5}}{2}\right)^{-1} + \left(\frac{1+\sqrt{5}}{2}\right)^{-2} \\
&= \frac{2}{1+\sqrt{5}} + \left(\frac{2}{1+\sqrt{5}}\right)^{2} \\
&= \frac{2}{1+\sqrt{5}} + \frac{4}{(1+\sqrt{5})^2} \\
&= \frac{2}{1+\sqrt{5}} + \frac{4}{(1+2\sqrt{5}+5)} \\
&= \frac{2}{1+\sqrt{5}} + \frac{4}{(6+2\sqrt{5})} \\
&= \frac{2}{1+\sqrt{5}} + \frac{4}{2(3+\sqrt{5})} \\
&= \frac{2}{1+\sqrt{5}} + \frac{2}{3+\sqrt{5}} && \text{Erweitern} \\
&= \frac{2}{1+\sqrt{5}} + \frac{2(3-\sqrt{5})}{(3+\sqrt{5})(3-\sqrt{5})} && \text{3. binomische Formel} \\
&= \frac{2}{1+\sqrt{5}} + \frac{2(3-\sqrt{5})}{9-5} \\
&= \frac{2}{1+\sqrt{5}} + \frac{2(3-\sqrt{5})}{4} \\
&= \frac{2}{1+\sqrt{5}} + \frac{3-\sqrt{5}}{2} && \text{Erweitern} \\
&= \frac{2(\sqrt{5}-1)}{(1+\sqrt{5})(\sqrt{5}-1)} + \frac{3-\sqrt{5}}{2} && \text{3. binomische Formel} \\
&= \frac{2(\sqrt{5}-1)}{5-1} + \frac{3-\sqrt{5}}{2} \\
&= \frac{2(\sqrt{5}-1)}{4} + \frac{3-\sqrt{5}}{2} \\
&= \frac{\sqrt{5}-1}{2} + \frac{3-\sqrt{5}}{2} \\
&= \frac{\sqrt{5}-1+3-\sqrt{5}}{2} \\
&= \frac{-1+3}{2} \\
&= \frac{2}{2} \\
&= 1
\end{aligned}
$$

Dies gilt in ähnlicher Weise auch für $b^{-1} + b^{-2}$.

Damit gilt:

$$
\begin{aligned}
f(n) &= \frac{1}{\sqrt{5}}(a^n \cdot 1 - b^n \cdot 1) && \text{Beweis siehe oben} \\
&= \frac{1}{\sqrt{5}}(a^n - b^n) && \text{Neut. Element} \\
&= \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right) && \text{Rücksubstitution}
\end{aligned}
$$

Es gilt somit $P(n)$.

8.5 Lemma 3:

Sei T ein AVL-Baum mit der Höhe n und sei $gb(T)$ die Anzahl der Ghostblätter von T. Dann gilt:
$gb(T) \geq F_n$

Beweis:
Induktion über n:

Induktionsanfang $h(T) = 0 \implies gb(T) = 1 \geq 0 = F_0$
$h(T) = 1 \implies gb(T) = 2 \geq 1 = F_1$

Induktionsschritt Sei $h(T) = n$. Dann ist im Worst-Case-Szenario (aufgrund der AVL-Eigenschaft), dass einer der Unterbäume (T') die Höhe $n-1$ und der andere(T") $n-2$ hat. Somit folgt
$gb(T) \geq gb(T') + gb(T") \overset{IV}{\geq} F_{n-1} + F_{n-2} = F_n$
Somit gilt die Aussage

Beweis für Lemma 1:
Zuerst zeigen wir, dass ein jeder AVL-Baum mit der Höhe h hat mindestens F_{h+1} ghosts.

Induktionsanfang Ein AVL-Baum der Höhe 0 hat einen ghost. Ein AVL-Baum der Höhe 1 hat 2 ghosts. Also: $h = 0 \implies 1 \geq F_{0+1} = F_1 = 1$
$h = 1 \implies 2 \geq F_2 = F_{1+1} = 1$

Induktionsschritt Sei T ein AVL-Baum der Höhe h. Seien T_1 und T_2 dessen Unterbäume. Ohne beschränkung der Allgemeinheit habe nun T_1 die Höhe $h-1$. Somit folgt aus der AVL-Eigenschaft, dass $h(T_2) \geq h-2$. Aus der Induktionsvoraussetzung folgt, dass T_1 mindestens F_h und T_2 mindestens F_{h-1} Blätter hat. Somit folgt, dass T mindestens $F_h + F_{h-1} = F_{h+1}$ Blätter

hat.

Die Aussage des Lemmas folgt dann aus:
Sei T ein AVL-Baum und n die Anzahl seiner realen Knoten (und somit $n+1$ die Anzahl and ghosts) und h die Höhe. Die Folgende ungleichung liefert die Aussage:
$n+1 \overset{Lemma3}{\geq} F_{h+1} \overset{Lemma2}{\geq} \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{h+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{h+1}\right) \geq \frac{1}{2\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+1}$

8.6 Insert

Wann immer wir einen Knoten zu einem AVL-Baum hinzufügen, wird dieser ein Blatt. Da wir allerdings ghosts an jedem Blatt hängen haben, wird zuerst ein ghost mit dem neuen Knoten der Höhe 1 ersetzt und dieser neue Knoten bekommt wieder zwei ghosts. Hiernach könnte allerdings die AVL-Eigenschaft nicht mehr gelten.

Betrachten wir y als unseren Knoten, den wir in unseren AVL-Baum T eingefügt haben. Offensichtlich kann die AVL-EIgenschaft nur auf dem Weg von der Wurzel zum Knoten y verletzt worden sein.
Im folgenden betrachten wir die rekursive Funktion $res(y)$ die die AVL-Eigenschaft des Baumes wiederherstellt (Anmerkung: y betietelt im folgenden nicht mehr den eingefügten Knoten,sondern das Argument auf das res aufgerufen wird.
Wenn y ein linker Sohn von seinem parent x ist, so wurde die Balance von x erhöht. Wenn y ein rechter Sohn von x ist, so wurde die Balance von x erniedrigt. Sollte durch das einfügen von y $bal(x) = 0$ werden, so sind wir fertig, da die Höhe von $T(x)$ sich nicht verändert hat.
Sollte nun aber $bal(x) \in -1, 1$ sein, so wurde die Höhe von $T(x)$ um 1 erhöht. In diesem Fall überprüfen wir x (also rufen $res(x)$ auf).
Nur bei $bal(x) \in -2, 2$ ist die AVL-Eigenschaft verletzt. Wir betrachten im Folgenden nur -2, da 2 komplett analog ist.
Wenn nun also $bal(x) = -2$ ist, so folgt, dass $y = r(x)$ und $bal(y) \in \{-1, 1\}$. Wir führen eine Fallunterscheidung:

$\underline{bal(y) = -1}$ In diesem Fall führen wir eine "Linksrotation"durch. Hierfür sei T_{lx} der linke Unterbaum von x, T_{ly} der linke Unterbaum von y und T_{ry} der rechte Unterbaum von y. Wir "rotieren"jetzt, indem y an die Stelle von x wandert, $x = l(y)$ wird, T_{lx} bleibt der linke Unterbaum von x, T_{ly} wird der rechte Unterbaum von x und T_{ry} beibt der rechte Unterbaum von y. Nach dieser Rotation liegen alle Balancen wieder in $-1, 0, 1$. Außerdem gilt $h(T(x))$ vor $insert$ ist die selbe wie $h(T(y))$ nach dem $insert$ weshalb es keine weiteren Verletzungen der AVL-Eigenschaft geben kann und wir somit fertig sind.

$\underline{bal(y) = 1}$ In diesem Fall müssen wir eine links-doppel-Rotation durchführen (quasi eine rechts-Rotation um y gefolgt von einer links-Rotation um x, siehe auch 8.6.2). Hierfür sei T_{xl} der linke Unterbaum von x, $l(y) = z$, T_{ry} der rechte Unterbaum von y, T_{lz} der linke Unterbaum von z und T_{rz} der rechte Unterbaum von z. Die links-doppel Rotation ordnet dann den Baum um, indem z an die Stelle von x wandert, mit $l(z) = x$ und $r(z) = y$ wobei $L(x) = T_{lx}$, $R(x) = T_{lz}$, $L(y) = T_{rz}$ und $R(y) = T_{ry}$. Nach dieser Rotation gilt, dass $h(z)$ gleich $h(x)$ vor

dem *insert* gilt und wir somit fertig sind.

Somit haben wir in beiden Fällen die AVL-Eigenschaft wieder hergestellt und können die Ausführung von *res* beenden.

8.6.1 Pseudocode für Insert-repair

```
1  Input: AVL−Baum T, Knoten y
2  x:= parent(y)
3  while x ≠ NULL do
4    if y = left(x) then
5      bal(x) := bal(x) + 1
6    else
7      bal(x) := bal(x) − 1
8    if bal(x) = 0 then
9      return
10   if bal(x) = −2 or bal(x) = 2 then
11     rotate
12     return
13   y := x; x := parent(y)
```

8.7 Delete

Wenn wir einen Knoten in einem AVL-Baum löschen können drei verschieden Fälle einterten.
Fall 1: Der Knoten ist ein Blatt, wir löschen den Knoten und setzen an seine Stelle ein ghost.
Fall 2: Der Knoten hat einen Sohn, in diesem Fall löschen wir den Knoten und schieben seinen Sohn an die Stelle (überbrücken des Knotens)
Fall 3: Der Knoten hat zwei Söhne. Hier löschen wir dann einfach den *Successor* und überschreiben den Knoten mit dem Inhalt des gelöschten *Successor*.
Sei also v nun der gelöschte Knoten. Wir nehmen an es gäbe keine Referenzen auf v aber $parent(v)$ zeigt noch immer auf den Parent von v vor dem Löschen. Nach dem Löschen rufen wir die Funktion AVL-delete-repair mit v auf. Generell funktioniert diese ähnlich zu AVL-insert-repair indem sie auch von v aus zu der Wurzel sich hocharbeitet.

8.7.1 Pseudocode für Delete-repair

```
1  Input: AVL−Baum T, Knoten v
2  x:= parent(v)
3  while x ≠ NULL do
```

```
4    if v = left(x) then
5      bal(x) := bal(x) − 1
6    else
7      bal(x) := bal(x) + 1
8    if bal(x) = 1 or bal(x) = −1 then
9      return
10   if bal(x) = −2 or bal(x) = 2 then
11     rotate
12   v := x; x := parent(v)
```

Betrachten wir den Fall $bal(x) = -2$ genauer (der andere Fall ist analog). In diesem Fall ist v ein linker Sohn von x. Sei y der rechte Sohn von x. Dann haben wir die folgenden drei Fälle:

$\underline{bal(y) = -1}$ Hier machen wir eine Linksrotation.

$\underline{bal(y) = 0}$ Auch hier machen wir eine Linksrotation. Anschließend kann abgebrochen werden.

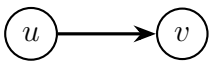
$\underline{bal(y) = 1}$ In diesem Fall führen wir eine doppelte Linksrotation durch.
Beachte: wir können nicht returnen nach dem Rotieren, da die Höhe sich durch das Löschen verändert hat.
Beide Repair-Funktionen gehen über einen Pfad der Länge $\mathcal{O}(logn)$. Jede Rotation braucht $\mathcal{O}(1)$, da wir eine konstante Zahl an Referenzen verändern.

9 Graph-Algorithmen

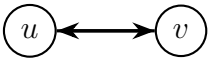
Allgemeine Notationen

Notation

Unter einem **Graphen** G versteht man ein Paar $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, wobei V der Knotenmenge und E der Kantenmenge entspricht. Man unterscheidet dabei

- **gerichtete Graphen**
  - V: i.d.R. $V = [1 : n]$
  - E: $E \subseteq V \times V$, $(u, v) \in E$: 

- **ungerichtete Graphen**
  - V: i.d.R. $V = [1 : n]$ (wie bei gerichteten Graphen)

– E: $\{u, v\} \in E$: 

Notation

Wir vereinbaren folgende Abkürzungen für verschiedene Algorithmen, die dazu dienen, einen Graphen zu durchsuchen:

- bfs: breadth first search
- dfs: depth first search

Beispiel

Sei M eine Menge. $M_k = \{m \subseteq M \mid \#m = 2\}$ entspricht der k-elementigen Teilmenge von M.

9.1 Datenstrukturen zur Repräsentation von Graphen

Definition

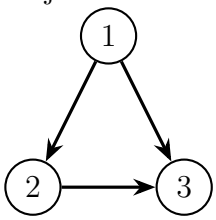
Unter einer **Adjazentmatrix** A_G eines Graphen G, $G = ([1 : u], E)$, versteht man eine $n \times n$-Matrix, für die folgende Eigenschaft gilt:

- $A_G[i, j] = 1 \leftrightarrow (i, j) \subseteq E$ (bzw. bei ungerichteten Graphen: $\{i, j\} \subseteq E$)
- $A_G[i, j] = 0$ sonst

Platz: $\mathcal{O}(n^2)$

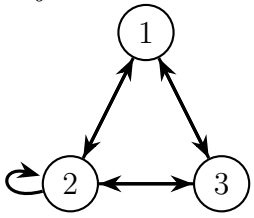
Beispiel

1. Adjazenzmatrix für gerichteten Graphen:



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

2. Adjazenzmatrix für ungerichteten Graphen:



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

Bemerkung

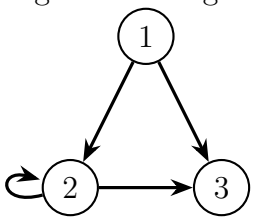
Die Adjazenzmatrix eines ungerichteten Graphen ist symmetrisch.

Definition

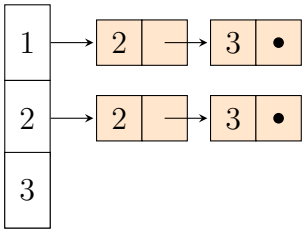
Gegeben sei ein Graph $G = (V, E)$, wobei $v \in V$. Unter einer **Adjazenzliste** $L(v)$ versteht man eine Liste mit Knoten w, wobei $(v, w) \in E$ (bzw. bei ungerichteten Graphen $\{v, w\} \in E$).

Beispiel

Gegeben sei folgender Graph:



Adjazenzliste zum Graphen:



9.2 Graph-Suche

Gegeben sei ein Graph $G = (V, E)$. Sei $s \in V$ der Startknoten. Ziel ist die Konstruktion eines Baumes mit allen von s erreichbaren Knoten, wobei s die Wurzel des Baumes sein wird.

Fragestellung: Existiert ein Pfad von s nach v, wobei $v \in V$, in G?

9.2.1 Breadth-First-Search

Sei $(v_0, v_1, ..., v_s)$ gegeben, wobei für alle i, $0 \leq i \leq s$, $(v_i, v_{i+1}) \in E$ (bzw. bei ungerichteten Graphen: $\{v_i, v_{i+1}\} \in E$) gilt. Sei $d(s, v)$ die Länge des kürzesten Pfades von s nach v. Seien $A \subset V$ die Menge der Knoten, von denen weiter gesucht wird, und $L(v)$ die Adjazenzliste zu v.
Lösung mit bfs: Konstruiert T (T = Menge von Kanten) und findet Knoten a in Reihenfolge ihrer Distanz $d(s, a)$.
Wir definieren eine Funktion λ, für die gilt:

- $\lambda(v) = 0$ falls v noch nicht besucht wurde (v ist new)
- $\lambda(v) = 1$ sonst (v ist old)

Wir speichen für alle $v \in V$ $\lambda(v)$ in einer Liste λ.
Initial: $T = \emptyset$, $\forall v \in V : \lambda(v) = 0$
Suchschritt: b-search
Pseudocode:

```
1  T = ∅ // global variable
2
3  bsearch(A) {
4      A' = ∅
5      forall v ∈ A{
6          forall u ∈ L(v){
7              if (λ(u) = 0){ //if new
8                  A' = A' ∪ {u}
9                  λ(u) = 1 //old
10                 T = T ∪ {(u, v)}
11             }
12         }
13     }
14     return A'
15 }
16
17 bfs(s) {
18     S_0 = s
19     i = 0
20     λ(s) = 1 //old
21     while S_i ≠ ∅{
22         S_{i+1} = bsearch(S_i)
23         i = i + 1
24     }
25 }
```

Lemma

$S_i = \{v \mid d(s, v) = i\}$

Beweis

Beweis durch Induktion über i.
Induktionsanfang $i = 0$

$d(s, v) = 0 \Rightarrow v = s$ (trivial)

Induktionsschritt $i \to i + 1$

$v \in S_{i+i} \cap L(u)$
$\Rightarrow d(s, u) = i$ (Induktionsvoraussetzung)
$\Rightarrow$ es existiert ein Pfad von s nach v mit Länge i, d(s,n) $\leq i + 1$
Annahme: $d(s, v) = j < i + i \Rightarrow d(s, v) = j \leq i + i$

$\Rightarrow v \in S_j$ (Induktionsvoraussetzung)
$\Rightarrow v$ war beim Aufruf von $bsearch(S_i)$ schon bekannt, $\lambda(v) = old$
$\Rightarrow v \notin S_{i+1}$
$\Rightarrow$ Widerspruch!

9.2.2 Depth-First-Search

Weiterhin gilt:

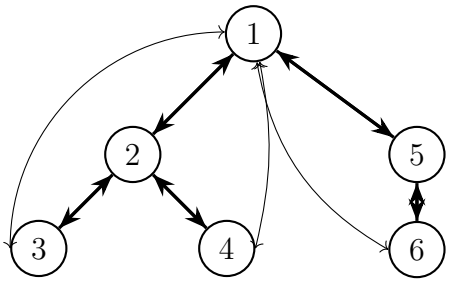
- $\lambda(v) = 0$ falls v noch nicht besucht wurde (v ist new)
- $\lambda(v) = 1$ sonst (v ist old)

Initial: $T = \emptyset$, $\forall v \in V : \lambda(v) = 0$
Pseudocode:

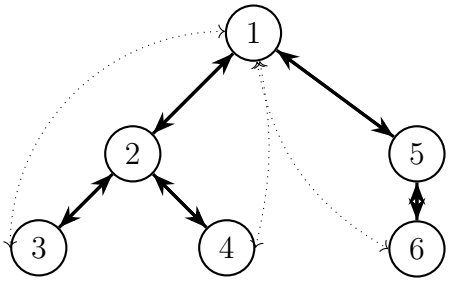
```
1  T = ∅ // global variable
2  dfs(v) {
3      if(λ(v) = 0) { // wenn neu
4          λ(v) = 1 // setze auf alt
5          forall u ∈ L(v) { // fuer alle Nachbarn
6                  if (λ(v) = 0) { // wenn neu
7                          T = T ∪ {(v,u) bzw. {v,u}} // fuege in den Baum ein
8                          dfs(u) // rekursiver Aufruf
9                  }
10         }
11     }
12 }
```

Beispiel

Betrache folgenden ungerichteten Graphen:



Spannbaum nach dfs():



Struktureigenschaften der dfs-Bäume:

- Tree-Edges →

- Back-Edges $\cdots$ (a,b)
  $\iff \exists$ ein Pfad von a nach b (in Richtung der Wurzel von den Blättern)
- Cross-Edges $\Longleftarrow$
  Können nur in gerichteten Graphen auftreten und dort auch nur von Knoten, weiter rechts im Strukturbaum, zu Knoten links im Strukturbaum

10 Potentialmethode

Notation

- (op_i): Folge von Rechenoperationen
- c_i: Kosten von op_i
- ϕ_i: Potenzial nach Operation i
- $\hat{c}_i$: Amortisierte Kosten von op_i

Definition

$\hat{c}_i = c_i + \phi_i$ - ϕ_{i-1}

Frage:
Was sind die Kosten einer Reihe von Operationen?

$$\sum_{i=1}^{n} c_i = ?$$

Aus unseren Definitionen folgt:

$$\begin{aligned} \sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \phi_i - \phi_{i-1}) \\ &= \sum_{i=1}^{n} (c_i) + \phi_n - \phi_0 \\ \Longrightarrow \sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} (\hat{c}_i) - \phi_n + \phi_0 \end{aligned}$$

Beispiel Binärer Zähler

k	...	2	1

Notation

- t_i: Länge des 1er Blocks im Zähler (beginnend am LSB)

Kosten einer Inkrementierung:

$$c_i = \begin{cases} t_i & \text{falls } t_i = \text{k} \\ t_i + 1 & \text{sonst} \end{cases}$$

Potentialfunktion:
ϕ_i: Anzahl der Einsen im Zähler (unabhängig der Position)

Beobachtung:

- $\phi_i > 0$: $\phi_i =$ $\underbrace{\phi_{i-1}}_{\text{Vorhandene Einsen}}$ - $\underbrace{t_i}_{\text{Überschriebener Block}}$ $+ 1$
- $\phi_i = 0$: $\phi_{i-1} = \text{k} = t_i$
  $\implies \phi_i \leq \underbrace{\phi_{i-1} - t_i}_{=0} + 1$

Es folgt:

$$\begin{aligned} \hat{c_i} &= c_i + \phi_i - \phi_{i-1} \\ &\leq t_i + 1 + \phi_{i-1} - t_i + 1 - \phi_{i-1} \\ &= 2 \end{aligned}$$

Nun können wir die Kosten abschätzen:

$$\begin{aligned} \sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} (\hat{c_i}) - \phi_n + \phi_0 \\ &\leq 2n + k \end{aligned}$$

11 Union-Find I

Notation

- Sei $\mathbb{S} = \{S_1, \ldots, S_k\}$ ein System disjunkter Mengen
- $r_i \in S_i$ = Repräsentant von S_i
  Wird im folgenden als Name der Menge dienen
- S(x) = S_i mit $x \in S_i$

Die Mengen werden wir als Bäume repräsentieren. Wobei jede Node einen Pointer auf seinen Parent besitzt. p(x) soll wie zuvor definiert sein.
Es soll gelten:

$$\text{x ist Repräsentant} \iff p(x) = x$$

Desweiteren soll r(x), Rang von x, eine obere Schranke für die Höhe des Baumes sein.

11.1 Operationen

1. **makeSet(x)**:
   Output: Erstellt neue Menge die nur aus x besteht und fügt sie dem Mengensystem hinzu. Das Mengensystem muss weiterhin disjunkt bleiben

```
1   makeSet(x) {
2       p(x) = x
3       r(x) = 0
4   }
```

2. **find(x) naiv**:
   Output: r_i mit $x \in \mathrm{S}(r_i)$

```
1   find(x) { //naiv
2       while p(x) != x {
3               x = p(x)
4       }
5       return x
6   }
```

3. **link(x,y)**:
   Output: Hängt den Baum\Node mit dem niedrigeren Rang in den anderen Baum\Node

```
1   if (r(x) < r(y)) {
2       p(x) = y
3   }
4   if (r(x) > r(y)) {
5       p(y) = x
6   }
7   if (r(x) = r(y)) {
8       p(x) = y
9       r(y) = r(y) + 1
10  }
```

4. **union(x,y)**:
   Output: $\mathbb{S}\backslash\{S(x), S(y)\} \cup \{S(x) \cup S(y)\}$
   Vereinigt die Menge $S_x = S(x)$, die x enthält mit der Menge $S_y = S(y)$, die y enthält

```
1    link(find(x), find(y))
```

Lemma

$$\begin{aligned} &|S(x)| = A \\ \implies &r(x) \leq \lfloor log_2(A) \rfloor \end{aligned}$$

Lemma

$$r(x) \geq h(x)$$

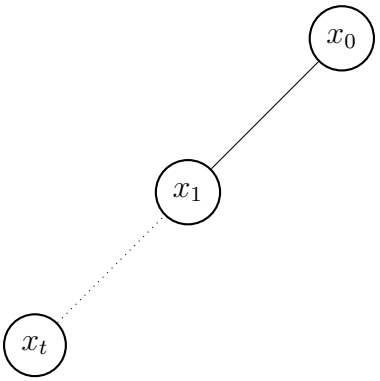
11.2 Kosten der Operationen

makeSet(x) $= \mathcal{O}(1)$
find(x) $= h(r_i)$ mit x $\in S_i \leq r(find(x)) \leq \lfloor log_2(n) \rfloor$
Union(x,y) $= \mathcal{O}(log_2(n))$

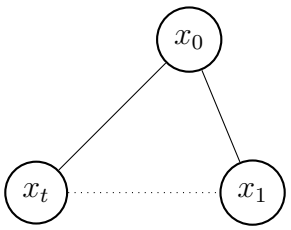
11.3 Path Compression

Nun soll **find(x)** nicht nur den Repräsentanten von x finden, sondern auch noch Path Compression betreiben

Vor Ausführung von find(x_t)



Nach Ausführung von find(x_t)



Es soll also gelten:

$$p(x_1) = \cdots = p(x_t) = x_0$$

find(x_t) mit Path Compression
Output: r_i mit $x_t \in \mathrm{S}(r_i)$
Seiteneffekt: $p(x_1) = \cdots = p(x_t) = r_i$

```
1 find(x):
2         if x != p(x) {
3             p(x) = find(p(x))
4         }
5         return p(x)
```

11.4 Korrektheit von find(x)

Beweis

Beweis per Induktion über die Tiefe von x

- Induktionsbasis:
  Tiefe(x) = 0 $\implies$ x = p(x)
  Damit ist der Basisfall bewiesen
- Induktionsschritt:
  t $\rightarrow$ t+1
  find(x_{t+1})
  $\implies$ $x_{t+1} \neq x_t$ $\implies$ rekursiver Aufruf von **find**(x_t)
  Per Induktionsvorraussetzung ist dieser korrekt und liefert den Output x_0
  $\implies$ p(x_{t+1}) = x_0
  Damit ist der Output von **find**(x_{t+1}) x_0, was zu Zeigen war.

PATH COMPRESSION OHNE REKURSIVER AUFRUF WAHRSCHEINLICH KLAUSURAUFGABE

11.5 Mathematische Probleme

Um 1900 Hilbert: “Alle mathematischen Probleme sind lösbar”

Beweisskizze
Formalisiere:

1. mathematische Aussagen A
2. mathematische Beweise B

Zeige: Rechenverfahren
Input: Aussage A
Output: $\begin{cases} 1 & \text{falls es } \exists \text{ Beweis B für A} \\ 2 & \text{falls es } \nexists \text{ Beweis B für A} \end{cases}$
$\Rightarrow$ evtl. lange Laufzeit.

Verdacht: Solches Verfahren gibt es nicht (Beweis: später Gödel).
Formal: $\forall$ Verfahren V: V kann es nicht.

$\Rightarrow$ Was ist ein Rechenverfahren?
Erster Ansatz: **primitiv rekursive** Funktionen (PR) $f : \mathbb{N} \to \mathbb{N}$.

$c(x) = c$	Konstante
$\prod_i^r(x) = x$	Projektion
$N(x) = x + 1$	Nachfolger
$f(x) = h(g_1(x), , ..., g_s(x))$	Einsetzen

primitiv rekursive Funktion mit einer Rekursionsvariablen:

$$f(0, x) = g(x)$$
$$f(n + 1, x) = h(n, x, f(n, x))$$

Beispiele:

- $f_1(0, x) = x$
  $f_1(n + 1, x) = N(f(n, x))$
  $\Rightarrow [f_1(n, x) = n + x]$

- $f_2(0, x) = 0$
  $f_2(n + 1, x) = f_1(n, f_2(n, x)) = x + f_2(n, x)$
  $\Rightarrow [f_2(n, x) = n \cdot x]$

- $f_3(0, x) = 1$
  $f_3(n + 1, x) = x \cdot f_3(n, x)$
  $\Rightarrow [f_3(n, x) = x^n]$

Vorgänger:

$$V(x) = \begin{cases} 0 & x = 0 \\ x - 1 & x > 0 \end{cases}$$

$$V(0) = 0$$
$$V(n + 1) = n$$

Ackermann: Es gibt berechenbare Funktionen, die nicht PR sind.
Idee: Mit k Anwendungen der PR-Regeln, kann man nur Funktionen mit begrenztem Wachstum erzeugen $f_k : \mathbb{N} \to \mathbb{N}$.

Iteration:
$f : \mathbb{N} \to \mathbb{N}$
$f^0(x) = x$
$f^{i+1}(x) = f(f^i(x))$

$$A_k(j) = \begin{cases} j+1 & k = 0 \\ A^j_{k-1}(j) & k > 0 \end{cases}$$

$A_0(j) = j + 1$
$A_1(j) = 2j$
$A_3(j) = 2^j \cdot j$

$$\begin{array}{ccc} \boldsymbol{A_0(0)} & A_0(1) & A_0(2) \\ A_1(0) & \boldsymbol{A_1(1)} & A_1(2) \\ A_2(0) & A_2(1) & \boldsymbol{A_2(2)} \end{array}$$

$$\Rightarrow A_j(j)$$

Diagonalisierung wächst schneller als jede PR - Funktion.

Cormen Leiserson Rivest

$$A_k(j) = \begin{cases} j+1 & k = 0 \\ A^{j+1}_{k-1}(j) & k > 0 \end{cases}$$

Lemma 0
Übung: monoton auf k und j.

Lemma 1

$$A_k(j) \leq A_k(j+1)$$
$$A_1(j) = 2j + 1$$

Lemma 2

$$A_2(j) = 2^{j+1}(j+1) - 1$$

$A_0(1) = 2$
$A_1(1) = 3$
$A_2(1) = 7$
$A_4(1) >> 10^{80}$ #Atome im bekannten Universum

Inverse Funktion

$$\alpha(n) = min\{k|A_k(j) \geq n\}$$

$$\alpha(n) = \begin{cases} 0 & 0 \leq n \leq 2 \\ 1 & n = 3 \\ 2 & 4 \leq n \leq 7 \\ 3 & 8 \leq n \leq 2047 \\ 4 & 2048 \leq n \leq A_1(n) \end{cases}$$

12 Union Find II

$makeset(x)$
$link(x, y)$
$find(x, y)$
$union(x, y) = link(find(x), find(y))$

n: # makeset Anfangs
m: $n+$ $\#union + \#find$
m': $n+$ $\#link + \#find$
$m' \leq m \leq 3m'$

Satz
Laufzeit von Union Find $\mathcal{O}(n + m \cdot \alpha(n))$

Kosten		
$\mathcal{O}(1)$	$makeset(x)$	$r(x) = 0$
$\mathcal{O}(1)$	$link(x, y)$	$r(y) < r(x)$
		$r(y) \geq r(x)$

$$r'(y) = \begin{cases} r(y) + 1 & r(x) = r(y) \\ r(y) & sonst \end{cases}$$

s	$find(x)$
$\uparrow$	

skaliere Einheit der Zeitmessung

Notation

Formeln, die für alle t gelten: t weglassen
Statt $\Phi_t = \sum_x \Phi_t(x)$ schreiben wir $\Phi = \sum_x \Phi(x)$

Mit $U_t \to U$ gilt:
Schritte t: op_t, c_t, r_t, l_t, i_t

Mit $U_{t+1} \to U_{t'}$ gilt:
Schritte $t + 1$: $op'_t, c'_t, r'_t, l'_t, i'_t$

Folglich ist $\hat{c}_{t+1} = c_{t+1} + \Phi_{t+1} - \Phi_t = \hat{c}' = c + hi' - \Phi$

Amortisierte Analyse: 1) Ranks 2) Hilfsfunktionen (*level l* , *iter i*) 3) $\hat{c} = \mathcal{O}(\alpha(n))$

12.1 Ranks

Lemma 4

BEWEIS IST KLAUSURAUFGABE

$$r(x) \leq r(p(x))$$
$$r(x) = r(p(x)) \Leftrightarrow x = p(x)$$

Lemma 5

BEWEIS IST KLAUSURAUFGABE

1.
$$r(x_{s-1}) < ... < r(x_i)$$

2.
$$r(p(x)) \leq r'(p'(x))$$
$$r(x) \leq r'(x)$$
$$x \neq p(x) \rightarrow r'(x) = r(x)$$

Lemma 6

$$r(x) \leq \lfloor log(n) \rfloor \leq n - 1$$

Potentialfunktion:

$$\Phi = \sum_x \Phi(x)$$
$$x = p(x) \vee r(x) = 0 \rightarrow \Phi(x) = r(x) \cdot \alpha(n)$$

12.2 Hilfsfunktionen

level l

$$l(x) = max\{k | A_k(r(x)) \leq r(p(x))\}$$

Lemma 21.1

$$r(x) \geq 1 : 0 \leq l(x) < \alpha(n)$$

Beweis

$$\begin{array}{rll} A_0(r(x)) & = r(x)+1 & \text{Definition von } A_0 \\ & \leq r(p(x)) & \text{Lemma 5} \end{array}$$

$$\begin{array}{rll} r(p(x)) & < n & \text{Lemma 6} \\ & \leq A_{\alpha(n)}(1) & \text{Definition von } \alpha(n) \\ & \leq A_{\alpha(n)}(r(x)) & \text{Monotonie von } A_k(j) \end{array}$$

iter i

$$i(x) = max\{i | A^i_{l(x)}(r(x)) \leq r(p(x))\}$$

Lemma 21.2

$$r(x) \geq 1 : 1 \leq i(x) < r(x)$$

Beweis

$$\begin{array}{rll} A^1_{l(x)}(r(x)) & = A_{l(x)}(r(x)) & \text{Definition Iteration} \\ & \leq r(p(x)) & \text{Definition } level\ l \end{array}$$

$$\begin{array}{rll} r(p(x)) & < A_{l(x)+1}(r(x)) & \text{Definition } level\ l \\ & \leq A^{r(x)+1}_{l(x)}(r(x)) & \text{Definition } A_k(j) \\ & \leq A_{\alpha(n)}(r(x)) & \text{Monotonie von } A_k(j) \end{array}$$

Bemerkung

$$\begin{array}{l} p(x) \neq x \wedge l(x) = l'(x) \Rightarrow i(x) = i'(x) \\ \qquad \downarrow \\ r'(x) = r(x) \end{array}$$

Potentialfunktion:

$$\Phi(x) = \begin{cases} \alpha(n) \cdot rx) & x = p(x) \vee r(x) = 0 \\ (\alpha(n) - l(x)) \cdot r(x) - i(x) & sonst \end{cases}$$

$$\Phi = \sum_x \Phi(x)$$

Lemma 8

$$0 \leq \Phi(x) \leq \alpha(n) \cdot r(n)$$

Lemma 9 $x \neq p(x)$, op: find oder link

1) $\Phi'(x) \leq \Phi(x)$

2) $r(x) \geq 1 \wedge (l'(x) \neq l(x) \vee i'(x) \neq i(x)) \implies \Phi'(x) \leq \Phi(x) - 1$

Beweis

$x \neq p(x) \underset{L5}{\implies} r'(x) = r(x)$
$r(x) = 0 \implies \Phi(x) = \Phi'(x) = 0$
$\underline{r(x) \geq 1}$: Fälle

1. $l(x) = l'(x) \implies i(x) \leq i'(x)$
   Subfälle: $i(x) = i'(x) \implies \Phi'(x) = \Phi(x)$
   $i(x) \leq i'(x) + 1 \implies \Phi'(x) \leq \Phi(x) - 1$
2. $l'(x) \geq l(x) + 1$

$$\begin{aligned}\Phi(x) - \Phi'(x) &= (\alpha(n) - l(x)) \cdot r(x) - i(x)) - ((\alpha(n) - l'(x)) \cdot r(x) - i'(x)) \\ &= (l'(x) - l(x)) \cdot r(x) + i'(x) - i(x) \\ &\geq 1 \cdot r(x) + 1 - r(x) \\ &= 1 \\ &\implies \Phi'(x) \leq \Phi(x) - 1\end{aligned}$$

Lemma 10 op_{t+1} makeset, $\hat{c}' = \mathcal{O}(1)$

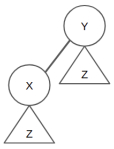
Beweis

$\Phi'(x) = 0$, $\Phi' = \Phi, c' = \mathcal{O}(1), \hat{c}' = c' + \Phi' - \Phi \implies c' = \mathcal{O}(1)$,

Lemma 11 *op* link(x, y), $\hat{c}' = \mathcal{O}(1) + \alpha(n)$

Beweis

OBdA:



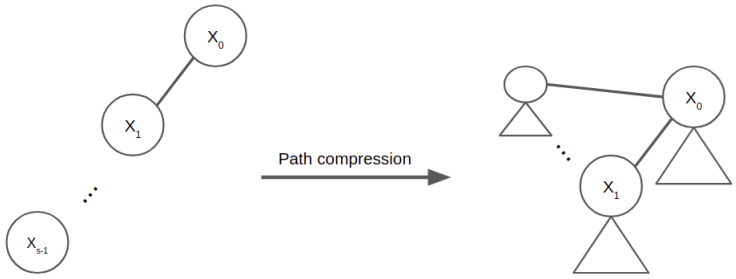
$\Phi'(z) \leq \Phi(z)$ (L9)

$r(x) = 0 \implies r'(x) = 0$, $\Phi'(x) = \Phi(x) = 0$

$$\begin{aligned}r(x) > 0 \implies \Phi'(x) &= (\alpha(n) - l'(x)) \cdot \underbrace{r'(x)}_{=r(x)} - i'(x) \\ &\leq \alpha(n) \cdot r(x) - 1 \\ &< \Phi(x)\end{aligned}$$

y: $\Phi(y) = r(y) \cdot \alpha(n)$, $p'(y) = y = p(y)$
$\Phi'(y) \in \{r(y) \cdot \alpha(n), (r(y) + 1) \cdot \alpha(n)\} \leq \Phi(y) + \alpha(n)$

Lemma 12 $op = \text{find}$, $\hat{i} = \mathcal{O}(\alpha(n))$ - Endgegner



$c \leq s$ (Einheit Zeitmessung)

$i \neq 0$: $\Phi'(x_i) \leq \Phi(x_i)$ (L9.1)
$\Phi'(x_0) = \Phi(x_0))r(x_0) \cdot \alpha(n)$
$x, y \in \{x_1, \ldots, x_{s-1}\}$ find path, nicht Wurzel

$E(x)$ Eigenschaft: $r(x) > 0 \wedge \exists y : p(y) \neq y, l(y) = l(x)$

$E(x)$ gilt möglicherweise nicht für:

1. $x = x_{s-1} \wedge r(x) = 0$
2. $x = x_0$
3. $\forall k \in [0, \alpha(n) - 1]$: letzter Knoten auf Pfad mit Level k

$\Longrightarrow \leq a(n) + 2$ viele Knoten.

E(x) gilt für $s - (\alpha(n) + 2)$ viele Knoten.

Behauptung: $E(x) \implies \Phi'(x) \leq \Phi(x) - 1$.

Beweis

Sei $k = l(x) = l(y)$.

(a) $r(p(x)) > A_k^{i(x)}(r(x))$ (Definition $i(x)$)

(b) $r(p(y)) > A_k^{(}r(y))$ (Def. $l(y)$)

(c) $r(y) \geq r(p(x))$ (L5.(a))

Sei $i = i(x)$

$$\begin{aligned} r(p(y)) &\geq A_K(r(y)) && \text{(b)} \\ &\geq A_K(r(p(y))) && \text{(c), Monotonie } A_K(...) \\ &\geq A_K(A_K^{i(x)}(r(x))) && \text{(a), Monotonie} \end{aligned}$$

$$
\begin{aligned}
r'(p'(x)) &= r(x_0) \\
&= r'(p'(y')) \\
&\geq r(p(y)) && \text{L5} \\
&\geq A_K^{i(x)+1} \underbrace{r(x)}_{=r'(x)} && \text{(d)}
\end{aligned}
$$

Falls $k = l'(x) : r'(p'(x)) \geq A_{l'(x)}^{i(x)+1}(r'(x)) \implies i'(x) \geq i(x)+1 \implies i(x) \neq i'(x)$

Falls $l \neq l'(x)$: Immer $l(x) = l'(x)$ oder $i(x) = i'(x)$ für x mit $E(x)$

$\Phi' \leq \Phi - (s - \underbrace{(\alpha(n)+2)}_{\leq \#\{x|E(x)\}})$. L9: $\Phi'(x) \leq \Phi(x) + 1$

$\hat{c}' = c' + \Phi' - \Phi \leq s - s + \alpha(n) + 2 = \mathcal{O}(\alpha(n))$

13 Pattern matching

Genom-Entzifferung: DNA-Schnipsel sukzessiv zu DNA durch pattern matching zusammensetzen.

Mit $x, y \in \Sigma^*$ (x, y sind also Strings / Zeichenketten) und $|b| = l$, finde erstes i, sodass $b = a[i : i+l-1]$. Definiere $test(i) = (b = a[i : i+l-1])$ mit den Kosten $\mathcal{O}(l)$.

Weiterhin sei x ein Suffix von y, wenn gilt: $\exists z : y = z \circ x$.

Naiver Algorithmus zum Pattern matching:

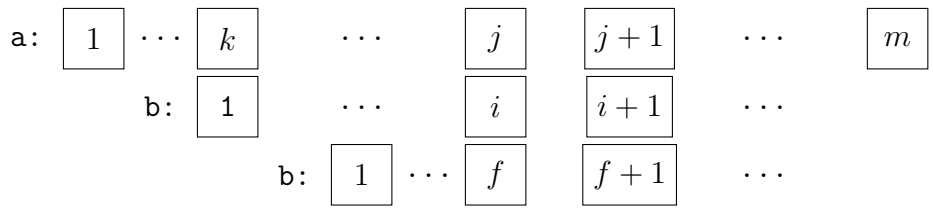
```
1 for i=1,...,m−l+1
2   test(i)
```

$\implies$ Laufzeit $\mathcal{O}((m-l) \cdot l) = \mathcal{O}(ml)$

Satz: $\mathcal{O}(l+m)$ möglich

13.1 Idee

Sei $a[k : j] = b[1 : i]$ und $b[1 : f]$ ein Suffix von b.



Wir testen ob $a_{j+1} = b_{i+1}$. Falls dies nicht der Fall ist(auch "failure"genannt) wird getestet ob $b_{f+1} = a_{j+1}$.

Definition

Wir definieren die failure-Funktion wie folgt:

$$f(i) = \begin{cases} max(\{S < i \mid b[1:S] = b[i-S+1:i]\}) & \text{falls existent} \\ 0 & \text{sonst} \end{cases}$$

Die iterierte failure-Funktion ist definiert als:

$$f^0(i) = i$$
$$f^{n+1}(i) = f(f^n(i))$$

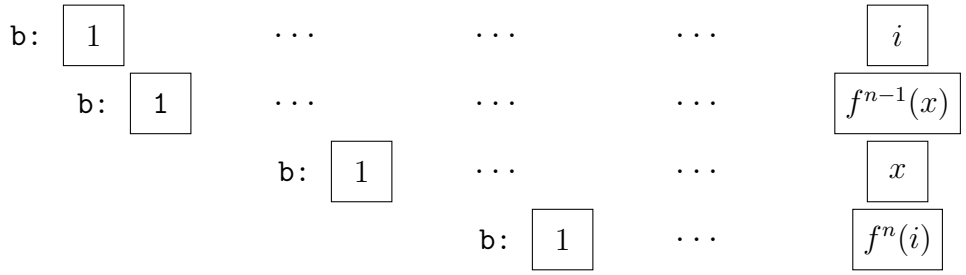
Lemma 1

$$\{b[1:f^m(i)] : n > 0\} = \{b[1:S] : b[1:S]\} \quad \text{(Suffix von } b[1:i])$$

Dies bedeutet, dass durch Iterieren von $f(i)$ alle Suffixe($a[1:S]$) von $a[1:i]$ erreicht werden.

Beweis

Sei $b[1:x]$ ein Suffix von $b[1:i]$.

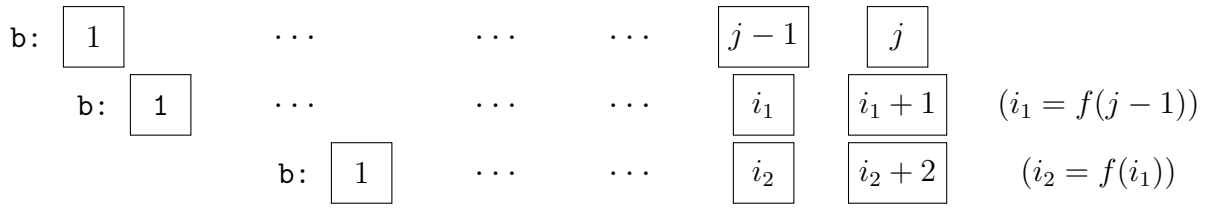


Aus $f(i) < i$ folgt nun:

$\exists n : f(i) < x \leq f^{n-1}(i)$
Falls $=$: ✓
Falls $<$:
$b[1:x]$ ist ein Suffix von $b[1:f^{n-1}(i)$ und $f(n) < x$
$\Rightarrow b[1...f(n)]$ ist nicht der längste Suffix von $b[1:f^{b-1}(i)]$
$\Rightarrow \lightning$

Berechnung der failure-Funktion $f(i)$ für alle i:

```
1   f(1) = 0;
2   for j = 2 to l {                    /* Schritt 1 (S1) */
3     i = f(j − 1);
4     while (b_j ≠ b_{i+1} and i > 0) {  /* Schritt 2 (S2) */
5         i = f(i);
6       }
7     if b_j ≠ b_{i+1} and i = 0 {       /* Schritt 3 (S3) */
8       f(j) = 0;
9     }
10    else {
11      f(j) = i + 1;
12    }
13  }
```



Die Korrektheit des Algorithmus folgt aus Lemma 1.

13.2 Laufzeit-Analyse

Laufzeit = O(Laufzeit für S1 + S2 + S3)

$T_j = \{t \mid \text{Schritt } t \text{ hat Typ S}j\}$

Kosten von Schritt t:

$C_t = 1$

$$\sum_t C_i = \sum_{j\in\{1,2,3\}} \sum_{t\in T_j} C_t$$
$$= \underbrace{\#T_1}_{l-1} + \#T_2 + \underbrace{\#T_3}_{l-1}$$

- Ausführen von S2 erniedridgt i
- Es gilt für jeden Schritt $i \geq 0$
- i wird höchstens $l-2$ mal erhöht (S3)

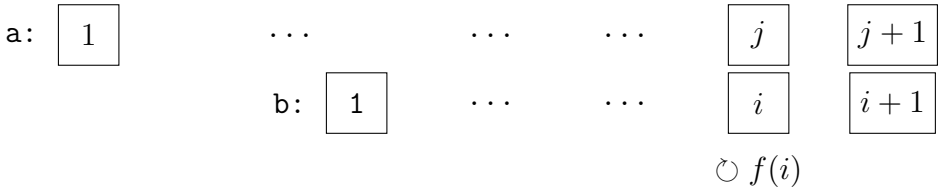
$\Rightarrow \#T_2 \leq l - 1$

$\Rightarrow$Laufzeit: $O(l)$

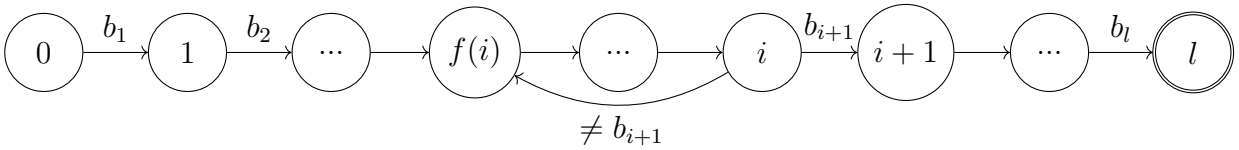
Laufzeit-Analyse anhand der Potenzial-Funktion
Gleiche Analyse durchführen mit $\phi = 1$ bzw. $\phi = i$ nach Schritt t und $C_t = 1$

ANALYSE IST WAHRSCHEINLICH KLAUSURAUFGABE

13.3 Pattern matching Algorithmus



Der Algorithmus kann ebenfalls als endlicher Automat dargestellt werden.



Inputstring: a_1, ..., a_m
Inputsymbole : τ

$$Z = \{0, \ldots, l\} \qquad \text{(Zustände)}$$
$$\delta : Z \times \Sigma \to Z$$
$$\delta(i, \tau) \begin{cases} i + 1 \text{ falls } \tau = b_i + 1 \\ f(i) \text{ sonst} \end{cases}$$
$$\Rightarrow \text{kann in } O(n) \text{ Schritten simuliert werden}$$

14 Kolmogorov-Komplexität

14.1 Einführung

Sei p ein Decoder für die Daten x, y:
p, x, y mit $p, x, y \in \{\mathbb{B}^*\}$

Das Problem, welches sich hierbei ergibt, ist, dass die Längen der einzelnen Komponenten p, x und y kodiert werden müssen. Um dies zu lösen wollen wir eine Kodierung finden, welche $x[n-1:0] \in \mathbb{B}^n$ als eine sich selbst begrenzende Zahl $x' \in \mathbb{B}^{n+x}$ darstellt.

Definition

Binärschreibweise ohne führende Nullen:

$$bin(n) \text{ mit n} \in \mathbb{N}$$

Kodierung:

$$h(0) = 00$$
$$h(1) = 11$$
$$h(y_1, \ldots, y_s) = h(y_1), \ldots, h(y_s)$$

Beispiel:

$$y = 110$$
$$h(y) = 111100$$

Definition

Sei $x \in \mathbb{B}^n$:

$$x' = \underbrace{h(bin(|x|)0010}_{O(\log n)}\underbrace{x}_{n})$$
$$\Rightarrow |x'| = n + O(\log(n))$$

Wir verwenden eine universelle Programmiersprache $L \in \mathbb{B}^*$. Wir verwenden hierfür den MIPS-Befehlssatz. Da dieser allerdings nur eine feste Registerbreite hat und damit lediglich so mächtig wie ein endlicher Automat ist, heben wir die Beschränkung der Registerbreite auf und erlauben beliebig breite Register. Alternativ könnten wir zur Implementierung auch eine Turingmaschine oder die Sprache C mit $range(int) = \mathbb{N}_0$ verwenden.

Notation:
$p \in L$ ist ein Programm. Zudem überladen wir die bisherige Notation und definieren $x \in \mathbb{B}^*$ als Input.

$$p(x) = \begin{cases} \text{Output von } p \text{ gestartet mit } x & \text{falls } p(x) \in \mathbb{B}^* \text{ und existiert} \\ \perp & \text{sonst} \end{cases}$$

Definition

d ist eine Beschreibung von y in L genau dann wenn $d = p'x$(Decoder gefolgt von Daten) und $p(x) = y$ mit $p \in L, x \in \mathbb{B}^*$.

Definition

$K_L(y) = min(\{|d| : d \text{ beschreibt } y\})$
$K_L(y)$ wird gesprochen als Kolmogorov-Komplexität von y. Wir merken an, dass K_L keine berechenbare Funktion ist.

Beispiel:
$p \in L$ mit Input x und Output x. $p'x$ beschreibt x bzw. $p(x) = x$. Da $p'x$ eine Beschreibung ist folgt:

$$\Rightarrow K_L(x) \leq |x| + \underbrace{|p'|}_{O(1)}$$

Anders ausgedrückt: Wir kodieren einen String durch sich selbst.

Beispiel:
$p \in L$ mit Input $bin(x)$ und Output $\underbrace{0, \ldots, 0}_{n}$. $p'bin(n)$ beschreibt also $\underbrace{0, \ldots, 0}_{n}$.

$$\Rightarrow K_L(\underbrace{0, \ldots, 0}_{n}) \leq \log(n) + O(1)$$

Bemerkung:
Die konkrete Programmiersprache ist nicht wichtig, da man einen Interpreter konstruieren kann, welcher eine Sprache A in eine Sprache B übersetzt. Die Abweichung wäre hierbei nur die Länge des Interpreters und daher irrelevant, aufgrund der O-Notation. Die Sprachen können dennoch nicht komplett beliebig gewählt werden, da die Sprachen mächtig genug sein müssen.

Definition

$x \in \mathbb{B}^n$ heißt K-zufällig, falls $K_L(x) \geq |x| = n$.

Lemma

$\forall n : \exists x \in \mathbb{B}^n : K_L(x) \geq n$

Bemerkung:

$$\begin{aligned}
\#\{x | K_L(x) < n\} &\leq \#\{d | d \text{ beschreibt ein Element aus } L\} \\
&\leq \#\mathbb{B}^1 + \#\mathbb{B}^2 + \ldots + \#\mathbb{B}^{n-1} \\
&= 1 + 2 + \ldots + 2^{n-1} \\
&= 2^n - 1 \\
&< 2^n
\end{aligned}$$

Ebenso gilt:

$$\begin{aligned}
\#\{x \in \mathbb{B}^n : K_L(x) < n - c\} &\leq \sum_{i=0}^{n-c-1} 2^i \\
&= 2^{n-c} - 1 \\
&< 2^{n-c}
\end{aligned}$$

Lemma 1

$$K_L(x) \leq |x| + O(1)$$

Lemma 2

$$\forall n \exists x \in \mathbb{B}^n : K_L(x) \geq n$$

Lemma 3

$$K_L \text{ nicht berechenbar}$$

Beweis

Programm p Annahme: doch

```
1  input: bin(n)
2  x = 0...0 ∈ B^n
3  while K(x) < n
4        {x = next_Lex (x) ∈ B^n}
5  output: x
```

Sei $x_n = \text{p(bin}(n))$ total wegen 2
$n \leq K_L(x_n) \leq |\text{p'bin}(n)| = O(1) + log(n)$ ↯

14.2 Bedingte Kolmogorov Komplexität

$$K_L(x|z) = min\{|p'y| : p \in L, P_L(z'y) = x\} \qquad \text{'gegeben z'}$$

Beispiel $x \in \mathbb{B}^n$ random

$$\overline{x}[1...n] = \overline{x_1}...\overline{x_n}$$
$$K(x|\overline{x}) = O(1)$$

Lemma 2'

$$\forall n, \forall y \exists x \in \mathbb{B}^n : K(x|y) \geq n$$

Lemma 4

$$\#\{x \in \mathbb{B}^n : K_L(x|y) < n - c\} < 2^{n-c}$$

Beweis

$$\#\{x \in \mathbb{B}^n : K_L(x|y) < n - c\} = \sum_{i=1}^{n-c-1} \#\mathbb{B}^i = \sum_{i=1}^{n-c-1} 2^i = 2^{n-c} - 1$$

Komprimierbare Strings sind selten:

$$\begin{aligned} x &\in \mathbb{B}^n \text{ durch Münzwurf} \\ P(x) &= \frac{1}{2^n} \\ \underset{Ereignis}{A \subseteq \mathbb{B}^n}\ P(A) &= \sum_{x \in A} p(x) = \frac{1}{2^n} \#A \end{aligned}$$

Lemma 5

$$P(\{x \in \mathbb{B}^n | K_L(x|y) < n - c\} \leq \frac{1}{2^n} \cdot 2^{n-c} = 2^{-c}$$

Komprimierbare Strings sind unwahrscheinlich

15 Random Routing (Valiant)

15.1 Einführung

Graph $G : (V, E)$ ungerichtet
$\pi : V \to V$
$\forall v \in V$: 1. Nachricht von v nach $\pi(v)$ senden
1. Schritt: über jede Kante ≤ 1 Nachricht schicken

Netzwerk: M-Cube

$$\begin{aligned} &\sigma_n = (\mathbb{B}^n, E_n) \\ &\{u, b\} \in E \Leftrightarrow \exists i : a_i \neq b_i \qquad \text{mit } u, b \in \mathbb{B}^n \\ &i = \dim(\{a, b\}) \end{aligned}$$

15.2 Satz(Valiant)

Geht (probabilistisch) mir großer Wahrscheinlichkeit in Zeit $O(n) = O(log(N))$
#Knoten= $\#\mathbb{B}^n = 2^n = N$
Idee 1: Würfle $\rho : \mathbb{B}^n \to \mathbb{B}^n$ $\underbrace{\rho(0...0)...\rho(1...1)}_{2^n}$ $\in \mathbb{B}^{n \cdot 2^n}$
$\overset{\longleftrightarrow}{n}$ $\overset{\longleftrightarrow}{n}$

Phase 1:
$\rho(v)$ Zwischenziel von Paket von Knoten v
sende Paket von v nach $\rho(v)$
über Pfad $v \to ... \to \rho(v)$ $v \to^* \rho(v)$
$d_1 < ... < d_s$ Dimensionen aufsteigend
$s \leq n$

Phase 2:
$\forall v$ sende paket von v nach $\rho(v)$ nach $\pi(v)$
Zeige für $i \in \{1, 2\}$ Stau in Phase unwahrscheinlich
Idee 2: Stau $\Rightarrow R$ komprimierbar $\overset{L5}{\to}$ ρ unwahrscheinlich
Stau:

$$\begin{array}{ccccccc} & & & & & & \rho(v) \\ & & & & & \nearrow & \\ n \to & ... & \overset{e(v)}{\to} & ... & \rho(n) & & \\ & & \nearrow & & & & \\ v \to & & & & & & \end{array}$$

$v \in \mathbb{B}^n$ $col(v) \Leftrightarrow \exists$ Kante $e(v)$, die von
$n \to^* \rho(n)$ und $v \to^* \rho(v)$ benutzt wird mit $e(v)$ Kollisionskante
$\dim(e(v)) = \dim(v)$ Kollisionsdimension

Sei $C = \sum c_i$ #Kollisionsknoten

$$c_i = \#\{v | col(v) \wedge \dim(v) = i\}$$
$$c_i \to c_i^* \quad \text{gerade} \quad c_i^* \geq c_i - 1$$

höchstens n Kanten, Routen entfernen, alle c_i^* gerade, von R gegen dim

A:

$$\underset{\leftarrow 2n \to}{LL\rho(n)}$$ extra: n für LL

B:

$$\underbrace{|...|}_{\frac{c_1^*}{2}} 0 \; ... \; 0 \underbrace{|...|}_{\frac{c_n^*}{2}} 0$$ extra: $\sum \frac{c_i^*}{2} + n = \frac{c}{2} + n$

$$c* = \sum c_i^* \leq c \cdot n$$

C:
Kollisionsknoten v_1 ... v_{c^*} nach dim(v) ordnen, c_i^* in dim i

$$\begin{array}{ccccc} \rho(v_1) & ... & \rho(v_i) & ... & \rho(v_{c^*}) \\ \text{ohne dim}(v_1) & & \text{ohne dim}(v_i) & & \text{ohne dim}(v_{c^*}) \end{array} \quad \textbf{gespart } c^*$$

D:
Restliche Start-Knoten

$$\begin{array}{rr} v_1 \ ... \ v_{2^n-c^*-1} & \text{ohne} n, v_i \\ \underset{\leftarrow n \rightarrow}{\rho(v_1)} \ ... \ \underset{\leftarrow n \rightarrow}{\rho(v_{2^n-c^*-1})} & \pm 0 \end{array}$$

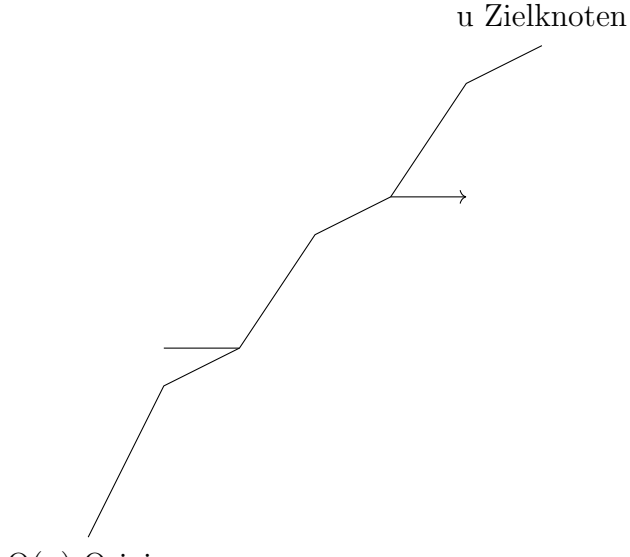
$$\begin{aligned} |ABCD| &= n2^n + 2n + \frac{c^*}{2} - c^* \\ &= n2^n + 2n - \frac{c^*}{2} \leq n2^n + 2n - \frac{c-n}{2} \\ |ABCD| &= n2^n + \frac{5n}{2} - \frac{c}{2} \\ K(\rho) &\leq n2^n + \frac{5n}{2} - \frac{c}{2} + O(1) \end{aligned}$$

Lemma 5

$$\begin{aligned} &P(\{\rho | lin\rho \leq c \text{ Kollisionen}\}) \ c = \frac{9n}{2} \\ &\leq 2^{-2n+O(1)} = 2^{O(1)} \frac{1}{(2^n)^2} = 2^{O(1)} \frac{1}{N^2} \end{aligned}$$

Betrachte statt $K(\rho|\text{dim}(n))$
$K(\rho|$ decodierer' dim$(n)) \rightarrow O(1)$ unnötig

Phase 2: Rückweg



Ordne immer nach Zielknoten $K(\rho|\text{decoder' lin}(n)'\pi)$
$\text{O}(s) = \text{O}(v) \rightarrow^* v = \pi(s)$ ergibt ρ