29.03.2018 – Vorlesung Randomisierte Algorithmen – Dietzfelbinger

1 Einführung und Beispiele

Die Vorlesung „Randomisierte Algorithmen“ befasst sich mit Algorithmen, die Zufallsexperimente durchführen. Dabei stellt man sich vor, dass dem Algorithmus bzw. dem ausführenden Rechner eine Operation „Wähle eine zufällige Zahl aus $\{1, \ldots, k\}$“ zur Verfügung steht, mit k als Parameter. Oft bestimmt dann die Eingabe x die Ausgabe $\mathcal{A}(x)$ von Algorithmus $\mathcal{A}$ auf x nicht mehr eindeutig; vielmehr ist dies eine Zufallsgröße. Von der Verwendung randomisierter Algorithmen verspricht man sich einen Effizienzgewinn. – Wir beginnen mit Beispielen für den Entwurf und die Analyse solcher Verfahren.

1.1 Randomisierter MinCut-Algorithmus

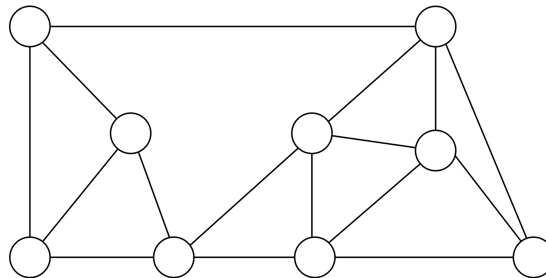

Abbildung 1.1.1: Ungerichteter zusammenhängender Graph $G = (V, E)$

In diesem Abschnitt betrachten wir zusammenhängende ungerichtete Graphen, bezeichnet mit $G = (V, E)$, s. Abb. 1.1.1 für ein Beispiel.

Definition 1.1.1. *Ein* ***Schnitt*** *in einem zusammenhängenden Graphen $G = (V, E)$ ist eine Kantenmenge $C \subseteq E$ derart, dass $(V, E - C)$ nicht (mehr) zusammenhängend ist. Ein Schnitt C heißt* ***minimal****, wenn es keinen Schnitt C' mit $|C'| < |C|$ gibt. (Schnitt minimaler Kardinalität.)*

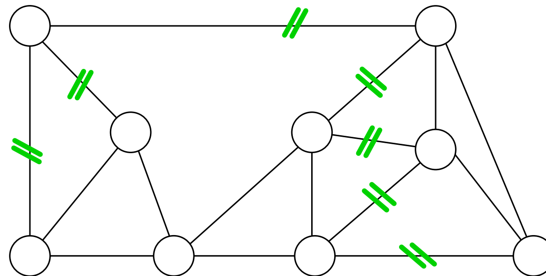

Abbildung 1.1.2: (Grün) Markierte Kanten: Schnitt C, nicht minimal

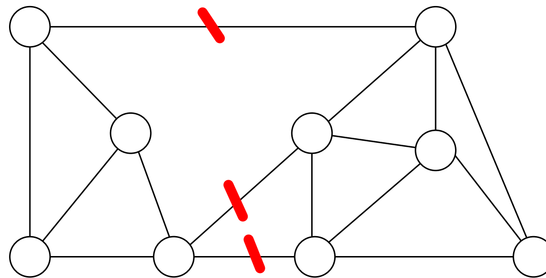

Abbildung 1.1.3: (Rot) Markierte Kanten: Schnitt C, minimal

In diesem Abschnitt betrachten wir die Aufgabe, in einem gegebenen Graphen einen minimalen Schnitt zu finden. Diese Aufgabe heißt das MinCut-Problem. In Abb. 1.1.2 ist ein nicht minimaler Schnitt eingezeichnet, in Abb. 1.1.3 ein minimaler Schnitt.

Bemerkung 1.1.2. Das MinCut-Problem besitzt effiziente Algorithmen, die ohne Randomisierung auskommen. Eine Sorte dieser Algorithmen beruhen auf Flussberechnungen (siehe „Effiziente Algorithmen" im Masterstudiengang) und haben Laufzeiten der Größenordnung $O(nm\log(n^2/m)) = O(nm\log n)$. Diese Algorithmen funktionieren sogar, wenn die Kanten mit Gewichten ≥ 0 versehen sind. Andere deterministische Algorithmen kommen ohne Flussberechnungen aus und haben Laufzeiten von $O(nm)$ [Stoer/Wagner 1997]. In [Brinkmeier 2007] wird ein Algorithmus vorgestellt, der Laufzeit $O(n^2\delta_G)$ hat, wobei δ_G der kleinste Knotengrad in G ist. (Beachte: $m \geq n\delta_G/2$,

also ist dies immer mindestens so gut wie $O(nm)$.)

Wir benutzen hier einen randomisierten Ansatz. Die Idee ist folgende. Wir bauen schrittweise einen Graphen (V, R) auf, beginnend mit $R = \emptyset$. Dazu wiederholen wir folgenden Schritt: Wähle zufällig eine Kante e aus E (und streiche e aus E). Wenn e zwei Zusammenhangskomponenten von (V, R) verbindet, füge e zu R hinzu. Dies wird iteriert, bis (V, R) genau zwei Zusammenhangskomponenten hat. Die Menge C aller Kanten zwischen diesen Zusammenhangskomponenten bildet einen Schnitt, der ausgegeben wird.

Wenn eine Kante gewählt wird, die innerhalb einer Zusammenhangskomponente von (V, R) verläuft, kann man diese Kante in der Analyse ignorieren. Jede Kante zwischen zwei Zusammenhangskomponenten hat dieselbe Wahrscheinlichkeit, als nächste gewählt zu werden. Man kann sich die Zusammenhangskomponenten von (V, R) auch so vorstellen: die Knotenmenge jeder Zusammenhangskomponente wird zu einem „Superknoten" zusammengefasst, und nur noch die Kanten werden betrachtet, die zwischen verschiedenen Superknoten verlaufen. Natürlich kann es passieren, dass zwischen Zusammenhangskomponenten mehrere Kanten verlaufen. Siehe Abb. 1.1.4.

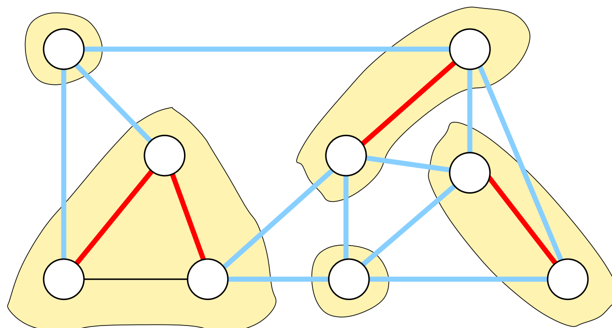

Abbildung 1.1.4: Rot: schon gewählte Kanten; blau: noch verfügbare Kanten; ocker: von den roten Kanten gebildete Zusammenhangskomponenten (Superknoten)

Eine *Runde* besteht nun in folgendem: Man wählt eine Kante aus denen zufällig, die zwischen verschiedenen Zusammenhangskomponenten verlaufen. Die gewählte Kante wird zu R hinzugefügt. Es werden $n-2$ Runden ausgeführt, bis nur zwei Zusammenhangskomponenten S und $V - S$ verbleiben. Die Ausgabe ist

$$C = \{(v, w) \mid v \in S,\, w \notin S\}.$$

Algorithmus 1.1.3 (BasicMinCut-Skizze).
INPUT: Ungerichteter zusammenhängender Graph $G = (V, E)$ mit $n = |V|$

METHODE:
1 Jeder Knoten in V bildet für sich eine Zusammenhangskomponente;
2 wiederhole $(n-2)$-mal:
3 wähle Kante (v, w) zufällig aus den Kanten, die zwei Zsh.-Komp. verbinden;
4 (vereinigt die beiden durch (v, w) verbundenen Zsh.-Komp. zu einer neuen;)
// es verbleiben 2 Zsh.-Komp. S und $V - S$
5 **return** C = Menge der Kanten aus E zwischen S und $V - S$.

Mit einer deutlichen Anlehnung an den Algorithmus von Kruskal können wir diese Skizze wie folgt implementieren. Wir benutzen eine Union-Find-Datenstruktur (siehe Vorlesung „Algorithmen und Datenstrukturen" im SS 2017). Die Details der Implementierung sind für das Verständnis der Wahrscheinlichkeitsanalyse des Algorithmus nicht wichtig. Die Menge R der gewählten Kanten muss man, anders als im Algorithmus von Kruskal, nicht mitführen.

Algorithmus 1.1.4 (BasicMinCut).
INPUT: Ungerichteter zusammenhängender Graph $G = (V, E)$ mit $n = |V|$
METHODE:
1 $[R \leftarrow \emptyset;]$
2 $F \leftarrow E$;
3 initialisiere eine Union-Find-Datenstruktur mit V als Objekten;
4 **while** in der U-F-Datenstruktur gibt es mehr als 2 Klassen **do**
5 wähle Kante (v, w) aus F zufällig; entferne (v, w) aus F;
6 $r \leftarrow \mathit{find}(v)$; $s \leftarrow \mathit{find}(w)$;
7 **if** $r \neq s$ **then** // (v, w) verbindet zwei Zsh.-Komp., neue Runde
8 [füge (v, w) zu R hinzu;] $\mathit{union}(s, t)$;
9 $C \leftarrow \emptyset$;
10 **for** Kante $(v, w) \in F$ tue:
11 **if** $\mathit{find}(v) \neq \mathit{find}(w)$ **then** füge (v, w) zu C hinzu;
12 **return** C.

Eine kurze Bemerkung zur Laufzeit: Wenn wir die Union-Find-Datenstruktur benutzen, die für $n - 2$ Union-Operationen Zeit $O(n \log n)$ benötigt und $O(1)$ für die Find-Operationen (Array-basiert, mit Listen für die Mengen), erhalten wir Laufzeit $O(n \log n + m)$ für den Algorithmus. Wenn die Kantenzahl nicht ganz klein ist, also

$m \geq n \log n$ gilt, ist dies ein *Linearzeitalgorithmus*. Alternativ können wir Union-Find mit wurzelgerichteten Bäumen implementieren, mit Union-by-Rank und Pfadkompression, und erhalten eine Laufzeit von $O(m \log^* n)$, auch für sehr dünne Graphen, also Kantenzahlen, die nahe bei n liegen. (Details: Vorlesung „Algorithmen und Datenstrukturen" im SS 2017.)

Nun wenden wir uns der Frage der Korrektheit zu. Gibt es Anlass zu glauben, dass der ausgegebene Schnitt C besonders wenige Kanten hat?

Definition 1.1.5. $\text{mc}(G) := \min\{|C| \mid C \text{ Schnitt in } G\}$*, die Größe eines minimalen Schnitts.*

Sei C_0 mit $|C_0| = k = \text{mc}(G)$ ein fest gewählter minimaler Schnitt. (Für ein Beispiel siehe Abb. 1.1.3. Beachte, dass C_0 nicht eindeutig bestimmt sein muss.) Wir werden abschätzen, mit welcher Wahrscheinlichkeit Algorithmus **BasicMinCut** genau C_0 ausgibt.

Wir beobachten: $(V, E - C_0)$ hat genau zwei Zusammenhangskomponenten. (Wenn es drei oder mehr gäbe, könnte man zu $E - C_0$ eine Kante aus C_0 hinzufügen, ohne dass der verbleibende Graph zusammenhängend wäre, im Widerspruch zur Minimalität von C_0.)

Wir nennen S_0 die Knotenmenge der einen und $\overline{S}_0 = V - S_0$ die Knotenmenge der anderen Komponente. Zudem sagen wir: ***Runde i*** in **BasicMinCut** ist der Schleifendurchlauf, in dem die $i$te Kante gefunden wird, die zwei Zusammenhangskomponenten verbindet. Es gibt also genau $n - 2$ Runden. Damit definieren wir $n - 2$ Ereignisse, die ausdrücken, dass in Runde i keine Kante aus C_0 gewählt wird.

$$\mathcal{E}_i := \{\text{die Kante, die in Runde } i \text{ gewählt wird, ist nicht in } C_0\}. \tag{1.1.1}$$

Lemma 1.1.6. $\mathbf{Pr}($*Algorithmus* ***BasicMinCut*** *liefert* C_0 *als Ausgabe*$) > 2/n^2$.

Beweis: Der Algorithmus gibt genau dann C_0 aus, wenn die beiden konstruierten Zusammenhangskomponenten genau S_0 und $V - S_0$ sind. Dies ist genau dann der Fall, wenn in keiner der $n - 2$ Runden eine Kante aus C_0 gewählt wird, und das heißt, dass alle $\mathcal{E}_i$, $1 \leq i \leq n - 2$, eintreten, formal: $\mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-2}$. Wir schätzen die Wahrscheinlichkeit hierfür nach unten ab.

Nach einer Grundformel der Wahrscheinlichkeitsrechnung gilt:

$$\mathbf{Pr}(\mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-2}) = \mathbf{Pr}(\mathcal{E}_1)\mathbf{Pr}(\mathcal{E}_2 \mid \mathcal{E}_1) \cdots \mathbf{Pr}(\mathcal{E}_{n-2} \mid \mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-3}). \tag{1.1.2}$$

(Bedingte Wahrscheinlichkeiten werden in Kapitel 2 wiederholt.) Wir müssen also

$$\mathbf{Pr}(\mathcal{E}_i \mid \mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{i-1})$$

abschätzen. Dazu nehmen wir an, dass in Runden $1, \ldots, i-1$ keine Kante aus C_0 gewählt wurde und dass nun Runde i stattfindet. Sei dazu E_{i-1} die Menge der Kanten nach Runde $i-1$, die zwei Zusammenhangskomponenten von (V, R) (Superknoten) verbinden. Jede dieser Kanten hat die gleiche Wahrscheinlichkeit, in Runde i gewählt zu werden.

Betrachte den Multigraphen[1] G_{i-1}, der aus den Superknoten nach Runde $i-1$ und E_{i-1} besteht. In diesem Graphen hat jeder Knoten Grad mindestens $k = \mathrm{mc}(G)$, weil die Menge aller Kanten, die aus einem Superknoten herausgehen, auf jeden Fall ein Schnitt in G ist. Da G_{i-1} genau $n-(i-1) = n-i+1$ Knoten hat, gilt

$$|E_{i-1}| \geq k(n-i+1)/2.$$

Die Wahrscheinlichkeit, in Runde i eine Kante aus C_0 zu wählen, ist also höchstens

$$\frac{|C_0|}{k(n-i+1)/2} = \frac{2}{n-i+1}.$$

Daher gilt:

$$\mathbf{Pr}(\mathcal{E}_i \mid \mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{i-1}) \geq 1 - \frac{2}{n-i+1} = \frac{n-i-1}{n-i+1}.$$

Nach (1.1.2) folgt

$$\mathbf{Pr}(\mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-2}) \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \frac{n-5}{n-3} \cdot \cdots \cdot \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3}. \qquad (1.1.3)$$

Kürzen liefert

$$\mathbf{Pr}(\mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-2}) \geq \frac{2}{n(n-1)} > \frac{2}{n^2}. \qquad \square$$

Die Wahrscheinlichkeit, dass $C \neq C_0$ ist, ist also höchstens $1 - 2/n^2$. Um die Fehlerwahrscheinlichkeit zu verringern, wiederholen wir Algorithmus **BasicMinCut** ℓ-mal und geben den kleinsten dabei produzierten Schnitt aus.

[1] In einem Multigraphen können zwischen zwei Knoten auch mehrere Kanten verlaufen.

Algorithmus 1.1.7 (MinCut).

INPUT: Ungerichteter zusammenhängender Graph $G = (V, E)$ mit $n = |V|$, $\ell \geq 1$.

METHODE:

1 $C \leftarrow \{(1, j) \mid (1, j) \in E\}$; // diese Kanten bilden einen Schnitt
2 $m \leftarrow |C|$;
3 **repeat** ℓ **times**
4 $\quad CC \leftarrow$ **BasicMinCut**(G);
5 $\quad$ **if** $|CC| < m$ **then**
6 $\quad\quad C \leftarrow CC$; $m \leftarrow |C|$;
7 **return** C.

Die Laufzeit dieses Algorithmus ist ℓ mal die Laufzeit von **BasicMinCut**. (Wir diskutieren weiter unten, wie ℓ zu wählen ist.) Es seien $C_1, \ldots, C_\ell$ die Ausgaben der ℓ Aufrufe, und es sei C^* die Ausgabe von **MinCut**. (Alle diese Kantenmengen sind Zufallsgrößen.) Wir analysieren: Wenn kein minimaler Schnitt ausgegeben wird, dann kann C_0 nicht in $\{C_1, \ldots, C_\ell\}$ vorkommen. Also:

$$\begin{aligned}
&\mathbf{Pr}(C^* \text{ ist kein minimaler Schnitt}) \\
&\quad \leq \mathbf{Pr}(C_0 \notin \{C_1, \ldots, C_\ell\}) \\
&\quad = \mathbf{Pr}(C_1 \neq C_0) \cdots \mathbf{Pr}(C_\ell \neq C_0) \\
&\quad \leq \left(1 - \frac{2}{n^2}\right)^\ell . \qquad (1.1.4)
\end{aligned}$$

(Beim Übergang von der zweiten zur dritten Zeile wurde die Unabhängigkeit benutzt, siehe Kap. 2.) Wenn wir z. B. $\ell = \lceil n^2/2 \rceil$ setzen, ergibt sich

$$\mathbf{Pr}(|C^*| > k) \leq \left(1 - \frac{2}{n^2}\right)^{n^2/2} < e^{-1} \approx 0.3679, \qquad (1.1.5)$$

also ist $\mathbf{Pr}(C^* \text{ ist minimaler Schnitt}) > 1 - e^{-1} > 0.632$. Wir haben hier die folgende wichtige Grundformel benutzt:

$$1 + z \leq e^z, \text{ für alle } z \in \mathbb{R}, \text{ Gleichheit genau für } z = 0.$$

Daraus folgt mit den Potenz-Rechenregeln

$$(1 + z)^y \leq e^{zy}, \text{ für alle } z \geq -1,\ y > 0.$$

Dies wurde für $z = -2/n^2$ und $y = n^2/2$ angewandt.

Wenn wir $\ell = \lceil c \cdot n^2 \cdot \ln(n)/2 \rceil$ setzen, für eine Konstante c, dann ergibt sich

$$\mathbf{Pr}(|C^*| > k) \le \left(1 - \frac{2}{n^2}\right)^{c \cdot n^2 \cdot \ln n/2} < e^{-c \ln n} = n^{-c}. \tag{1.1.6}$$

Mit einer relativ kleinen Wiederholungszahl lässt sich also die Fehlerwahrscheinlichkeit auf einen „polynomiell kleinen" Wert drücken.

Man beachte, dass die Gesamtlaufzeit des Algorithmus **MinCut** auf G mit n Knoten und mit $\ell = \lceil c \cdot n^2 \cdot \ln(n)/2 \rceil$ durch $O((n \log n + m) \cdot c \cdot n^2 \cdot \ln(n)) = O(n^3(\log n)^2 + mn^2 \log n)$ beschränkt ist. Auf jeden Fall ist die Laufzeit polynomiell.

Satz 1.1.8. *Algorithmus 1.1.7* ***MinCut*** *mit* $\ell = n^2/2$ $[\ell = n^2 \cdot \ln(n)/2;\ \ell = c \cdot n^2 \cdot \ln(n)/2]$ *liefert mit Wahrscheinlichkeit größer als* $1 - 1/e > 0.632$ $[1 - 1/n;\ 1 - 1/n^c]$ *einen Schnitt minimaler Kardinalität. Der Algorithmus hat in jedem Fall polynomielle Laufzeit.*

Mitteilung: Man kann den Algorithmus **MinCut** so modifizieren und implementieren, dass die Laufzeit geringer wird als für sämtliche bekannten deterministischen Algorithmen. Dies wurde von Karger und Stein 1993 geleistet. Ihr Algorithmus hat eine Laufzeit von $O(n^2(\log n)^4)$.

Bemerkung: Das MinCut-Problem hat also deterministische Polynomialzeitalgorithmen und einen etwas schnelleren randomisierten Algorithmus. In der Vorlesung „Automaten, Sprachen und Komplexität" lernt man die NP-vollständigen Probleme kennen, mit der Vermutung, dass es für sie *keinen* deterministischen Polynomialzeitalgorithmus gibt. Kann Randomisierung hier helfen? Leider kennt man kein einziges NP-schweres Problem, das einen randomisierten Polynomialzeitalgorithmus zulässt. Aufgrund von Resultaten in der Komplexitätstheorie gibt es sogar Anlass zu der Vermutung, dass es solche Probleme nicht gibt. Der Ansatz „Randomisierung" kann also zu effizienteren (d. h. schnelleren) Lösungen führen; vermutlich wird er aber nicht dazu führen, dass man NP-schwere Probleme effizient lösen kann. In der Vorlesung „Approximationsalgorithmen" kann man lernen (Master-Studium!), wie Randomisierung in Verbindung mit anderen Techniken dabei hilft, näherungsweise Lösungen für NP-schwere Probleme zu berechnen.

1.2 Gleichheit von Polynomprodukten

Beispiel: Gilt

$$(x^4+4x^3+6x^2+4x+1)(x^4-4x^3+6x^2-4x+1) = (x^2-1)(x^2-1)(x^2-1)(x^2-1)\ ? \tag{1.2.7}$$

Dabei ist natürlich die Gleichheit zwischen Polynomen gemeint. Allgemeiner könnte man $(\sum_i \binom{n}{i} x^i)(\sum_i (-1)^i \binom{n}{i} x^i)$ mit dem Produkt von n Kopien von (x^2-1) vergleichen.

Allgemein: Gegeben seien Polynome $f_1(x), \ldots, f_s(x)$ und $g_1(x), \ldots, g_t(x)$ über einem Körper K (etwa $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$ oder $ints_p$ für eine Primzahl p).

Frage: Gilt

$$f_1(x) \cdot \ldots \cdot f_s(x) = g_1(x) \cdot \ldots \cdot g_t(x)\ ? \tag{1.2.8}$$

Die naheliegende Methode ist, die Polynome auf beiden Seiten auszumultiplizieren, zu vereinfachen und dann zu vergleichen. Dabei beobachtet man folgendes: Die Multiplikation zweier Polynome vom Grad n benötigt bei naivem Vorgehen $\Theta(n^2)$ Körperoperationen. Mit der FFT-Methode (siehe Vorlesung „Algorithmen und Datenstrukturen") erreicht man für manche Bereiche (z. B. $\mathbb{R}$ oder $\mathbb{C}$) Kosten von $O(n \log n)$. Ähnliche Kosten ergeben sich bei der Multiplikation von n Kopien eines Polynoms vom Grad 2. Es ist kein deterministischer Algorithmus bekannt, der für solche Multiplikationsaufgaben mit linearem Aufwand auskommt.

Wir versuchen es mit folgendem Trick: Wir wählen ein zufälliges Argument, setzen es in die einzelnen Polynome ein und werten diese aus, multiplizieren dann die Werte und vergleichen die Produkte. (Damit werden keine Polynome mehr multipliziert.)

Dies heißt genauer: Sei z. B. K gleich $\mathbb{Q}$ oder eine Obermenge davon. Wähle a aus $\{1, \ldots, k\}$ rein zufällig, berechne $b_1 = f_1(a), \ldots, b_s = f_s(a), c_1 = g_1(a), \ldots, c_t = g_t(a)$ durch Einsetzen und Auswerten, und berechne dann $u = b_1 \cdots b_s$ und $v = c_1 \cdots c_t$. Falls $u \neq v$, weiß man sicher, dass die Produktpolynome verschieden sind. Falls $u = v$, muss man mit der Interpretation des Ergebnisses etwas vorsichtiger sein.

Wesentlich ist die Betrachtung der Grade. Sei

$$d := \max\Big\{ \sum_{1\le i\le s} \deg(f_i(x)), \sum_{1\le j\le t} \deg(g_j(x)) \Big\}.$$

Dann hat das Polynom

$$h(x) := f_1(x) \cdot \ldots \cdot f_s(x) - g_1(x) \cdot \ldots \cdot g_t(x) \tag{1.2.9}$$

auf keinen Fall einen Grad größer als d. Offensichtlich gilt für alle a:

$$h(a) = 0 \quad \Leftrightarrow \quad f_1(a) \cdot \ldots \cdot f_s(a) = g_1(a) \cdot \ldots \cdot g_t(a). \tag{1.2.10}$$

Bei der angedeuteten Berechnung erhalten wir also $u = v$ genau dann wenn $h(a) = 0$ ist. Wenn die Polynomprodukte gleich sind, ist $h(x)$ das Nullpolynom, und es gilt $h(a) = 0$ für alle a. Andernfalls ist $h(x)$ nicht das Nullpolynom.

Fakt 1.2.1. *Ein Polynom vom Grad höchstens d über einem beliebigen Körper (etwa $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, $\mathbb{Z}_p$ für eine Primzahl p), das nicht das Nullpolynom ist, hat in diesem Körper nicht mehr als d Nullstellen.*

Daher hat $h(x)$ nicht mehr als d Nullstellen in $\{1, \ldots, k\}$. Wenn wir $k \geq d$ wählen, erhalten wir für zufällig aus $\{1, \ldots, k\}$ gewähltes a:

$$\mathbf{Pr}(h(a) = 0) \leq \frac{|\{a \in \{1, \ldots, k\} \mid h(a) = 0\}|}{k} \leq \frac{d}{k}.$$

Durch geeignete Wahl von k können wir diese Wahrscheinlichkeit klein machen.

Algorithmus 1.2.2 (PPV – Polynomproduktvergleich).
Input: Listen $f_1(x), \ldots, f_s(x)$ und $g_1(x), \ldots, g_t(x)$ von Polynomen
Methode:
(1) $d \leftarrow \max\{\sum_{1\leq i\leq s} \deg(f_i(x)), \sum_{1\leq j\leq t} \deg(g_j(x))\}$;
(2) Setze $k \leftarrow 2d$;
(3) Wähle a aus $\{1, \ldots, k\}$ *zufällig* // (uniforme Verteilung).
(4) $u \leftarrow f_1(a) \cdot \ldots \cdot f_s(a)$;
(5) $v \leftarrow g_1(a) \cdot \ldots \cdot g_t(a)$;
(6) **if** $u \neq v$ **then return** 1
(7) **else return** 0.

Die zentralen Eigenschaften von Algorithmus 1.2.2 sind folgende:

- Die Anzahl der Multiplikationen von Körperelementen ist höchstens $4d + s + t$, die Rechenzeit ist $O(d)$, wenn man die einzelnen Polynome mit Hilfe des Horner-Schemas auswertet.
- Wenn $f_1(x) \cdot \ldots \cdot f_s(x) = g_1(x) \cdot \ldots \cdot g_t(x)$, ist die Ausgabe sicher 0.
- Wenn $f_1(x) \cdot \ldots \cdot f_s(x) \neq g_1(x) \cdot \ldots \cdot g_t(x)$, gilt: $\mathbf{Pr}(\text{Ausgabe ist } 0) \leq \frac{1}{2}$ (oder, bei anderer Wahl von k im Algorithmus: $\ldots \leq d/k$).

Wenn man höhere Zuverlässigkeit möchte, könnte man viel größere k wählen. Für Körper, bei denen eine Rechenoperation tatsächlich mit Zeit $O(1)$ zu veranschlagen ist, ist dies ohne große Laufzeitveränderung möglich. Schwierigkeiten entstehen, wenn der Körper nur endlich viele Elemente hat oder wenn, wie im Fall $K = \mathbb{Q}$, Rechnen mit größeren Zahlen größeren Aufwand hat. Alternativ führt man Algorithmus 1.2.2 mehrfach (ℓ-mal) durch, mit Ergebnissen $b_1, \ldots, b_\ell$, und liefert $\max\{b_1, \ldots, b_\ell\}$ als Gesamtergebnis. Für diesen Algorithmus gilt:

- Die Anzahl der Multiplikationen von reellen Zahlen ist höchstens $(4d + s + t)\ell$.
- Wenn $f_1(x) \cdot \ldots \cdot f_s(x) = g_1(x) \cdot \ldots \cdot g_t(x)$, ist die Ausgabe sicher 0.
- Wenn $f_1(x) \cdot \ldots \cdot f_s(x) \neq g_1(x) \cdot \ldots \cdot g_t(x)$, gilt: $\mathbf{Pr}(\text{Ausgabe ist } 0) \leq \frac{1}{2^\ell}$ (oder, bei anderer Wahl von k im Algorithmus: $\ldots \leq (d/k)^\ell$).

Als Zahlenbeispiel nehmen wir $d = 10\,000$ und $\ell = 100$ an. Die Anzahl der Multiplikationen ist dann höchstens $4 \cdot 10^6$, was sich in Sekundenbruchteilen berechnen lässt. Die Wahrscheinlichkeit, ein falsches Resultat zu erhalten, ist nicht größer als

$$\frac{1}{2^{100}} = \frac{1}{(2^{10})^{10}} < \frac{1}{1000^{10}} = \frac{1}{10^{30}}.$$

Diese Wahrscheinlichkeit ist (zum Beispiel) viel kleiner als die dafür, dass der Rechner während der Rechnung durch einen Hardwarefehler ausfällt. (Zum Vergleich halte man sich vor Augen, dass die Anzahl der Nanosekunden in 20 Jahren weniger als 10^{18} ist!) Aus Anwendungssicht ist der Algorithmus also ebenso gut wie ein deterministischer Algorithmus.

Wenn man Polynommultiplikation naiv implementiert, müsste man mit etwa $d^2 = 10^8$ Multiplikationen rechnen, was viel mehr als $4 \cdot 10^6$ ist. FFT kann in manchen Fällen Abhilfe schaffen. Im Kapitel über „Algebraische Methoden“ werden wir Produktvergleich für multivariate Polynome betrachten, bei der es keine bekannten schnellen deterministischen Algorithmen gibt.

1.3 Verifikation von Matrixprodukten

Gegeben seien drei $n \times n$-Matrizen A, B und C über einem Körper K (zum Beispiel $K = \mathbb{Q}$ oder $K = \mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ mit Operationen modulo p, wobei p eine Primzahl ist). Jemand behauptet, dass $A \cdot B = C$ ist. Wir wollen dies überprüfen.

Eine naheliegende Methode ist, das Produkt $A \cdot B$ zu berechnen und das Resultat mit C zu vergleichen. Mit der Schulmethode für die Multiplikation dauert dies Zeit $O(n^3)$, mit der Strassen-Methode (Vorlesung „Algorithmen und Datenstrukturen") $O(n^{\log 7})$, wobei $\log 7 \approx 2.81$.[2] Wir zeigen hier, dass ein sehr einfacher randomisierter Algorithmus mit Zeit $O(n^2)$ auskommt.

Algorithmus 1.3.1 (VerifyMatrixProduct).
Input: $n \times n$-Matrizen A, B, C über Körper K.

(1) Wähle endliche Menge $S \subseteq K$ mit $0, 1 \in S$;
(2) Wähle Vektor $v = (v_1, \ldots, v_n) \in S^n$ *zufällig* // (uniforme Verteilung);
(3) $u \leftarrow B \cdot v$;
(4) $w \leftarrow A \cdot u$;
(5) $x \leftarrow C \cdot v$;
(6) **if** $w \neq x$ **then return** 1 **else return** 0.

Die Laufzeit dieses Algorithmus ist $O(n^2)$, weil drei Matrix-Vektor-Multiplikationen auszuführen sind.

Wie steht es mit dem Ein-Ausgabe-Verhalten? Offensichtlich gilt folgendes: Wenn $A \cdot B = C$ ist, dann gilt

$$x = C \cdot v = (A \cdot B) \cdot v = A \cdot (B \cdot v) = A \cdot u = w,$$

also ist die Ausgabe 0. (In der Rechnung haben wir die Assoziativität im Produkt $A \cdot B \cdot v$ benutzt: der Trick ist, dass wir auf diese Weise mit $A \cdot B$ hantieren können, ohne dieses Matrixprodukt auszurechnen!)

Ab hier gelte $A \cdot B \neq C$. Wir definieren

$$D := A \cdot B - C,$$

mit $D = (d_{ij})_{1 \leq i,j \leq n}$. Weil $A \cdot B \neq C$, gibt es einen Eintrag $d_{i_0 j_0} \neq 0$. Wir überlegen: Wenn $v_1, \ldots, v_{j_0-1}, v_{j_0+1}, \ldots, v_n$ irgendwelche Elemente von S sind, dann gibt es in S entweder ein oder kein Element v_{j_0} mit

$$d_{i_0 1} v_1 + \cdots + d_{i_0, j_0-1} v_{j_0-1} + d_{i_0 j_0} v_{j_0} + d_{i_0, j_0+1} v_{j_0+1} + \cdots + d_{i_0 n} v_n = 0. \qquad (1.3.11)$$

[2] Schnellere Methoden: Coppersmith und Winograd (1987/90), Zeit $O(n^{2.376})$; Stothers (2010), Williams (2011), Le Gall (2014): Zeit $O(n^{2.3728639})$. (Diese Algorithmen gelten allerdings als rein theoretisch wichtig; für realistisch große n sind andere Verfahren schneller.)

Dies liegt daran, dass man wegen $d_{i_0 j_0} \neq 0$ die Gleichung (1.3.11) nach v_{j_0} auflösen kann. Eine Lösung gibt es, wenn das Ergebnis in S liegt, keine Lösung sonst. Damit erhalten wir: Wenn v aus S^n zufällig gewählt wird, ist die Wahrscheinlichkeit dafür, dass (1.3.11) gilt, höchstens $1/|S|$.

Wir können also abschätzen:

$$\begin{aligned}
&\mathbf{Pr}(\text{Ausgabe ist } 0)\\
&= \mathbf{Pr}(A \cdot B \cdot v = C \cdot v) \qquad (\text{für den Zufallsvektor } v)\\
&= \mathbf{Pr}(D \cdot v = 0)\\
&\leq \mathbf{Pr}(\text{Gleichung (1.3.11) gilt})\\
&\leq \frac{1}{|S|} \leq \frac{1}{2}.
\end{aligned} \tag{1.3.12}$$

Wir fassen zusammen: Algorithmus 1.3.1 hat folgendes Verhalten: Wenn $A \cdot B = C$ ist, ist die Ausgabe 0; wenn $A \cdot B \neq C$ ist, entsteht die (unerwünschte) Ausgabe 0 höchstens mit Wahrscheinlichkeit $1/|S|$.

Spezialfall: Wenn $K = \mathbb{Z}_p$ für eine Primzahl p, wird man $S = \mathbb{Z}_p$ wählen; die Fehlerwahrscheinlichkeit ist dann (genau) $1/p$.

Wenn die Fehlerwahrscheinlichkeit $1/|S|$ zu groß ist, können wir ebenso wie in Abschnitt 1.1 durch ℓ-fache Wiederholung die Fehlerwahrscheinlichkeit auf $(1/|S|)^\ell$ drücken. Beispielsweise genügt $(c \log n)$-fache Wiederholung, um eine Fehlerwahrscheinlichkeit von höchstens $|S|^{-c \log n} = n^{-c \cdot \log |S|}$ zu erhalten. Dann hat der Algorithmus eine Laufzeit von $O(n^2 \log n)$, immer noch viel weniger als die deterministischen Verfahren.

1.4 Randomisiertes Quicksort

Der Algorithmus „Quicksort“ wurde schon in mehreren Vorlesungen betrachtet: „Algorithmen und Programmierung“ und „Algorithmen und Datenstrukturen“. Hier konzentrieren wir uns auf die „randomisierte“ Variante und auf eine alternative Analysemethode, die einige interessante Ansätze benutzt.

Wir gehen von folgender idealisierter Situation aus: In einem Array `A[`$1..n$`]` stehen verschiedene Zahlen $a_1, \ldots, a_n$, in dieser Reihenfolge. Die Aufgabe ist, diese Zahlen umzusortieren, in eine Reihenfolge $b_1 < \cdots < b_n$, wobei b_i in `A[`i`]` steht, für $1 \leq i \leq n$.

Randomisiertes Quicksort geht so vor: Wenn $n = 1$ ist, passiert nichts. Wenn $n > 1$ ist, wird eine Prozedur **rqsort**$(1, n)$ aufgerufen. Dabei ist **rqsort**(l, r) so angelegt, dass gilt: Die Einträge in `A`$[l..r]$ werden sortiert, die Einträge in `A`$[1..l-1]$ und `A`$[r+1..n]$ bleiben unverändert. Die Prozedur **rqsort**(l, r) geht wie folgt vor: Wenn $l \geq r$ gilt, passiert nichts. Wenn $l < r$ gilt, wird ein Index $t \in \{l, \ldots, r\}$ ***zufällig*** gewählt. Der Eintrag x in `A`$[t]$ wird das „Pivotelement" oder „partitionierende Element". Durch eine Prozedur „**partition**" werden die Einträge in `A`$[l..r]$ wie folgt verschoben: Der Eintrag x landet in einer Position `A`$[p]$ mit $l \leq p \leq r$, alle Einträge in `A`$[l..p-1]$ sind kleiner als x, alle Einträge in `A`$[p+1..r]$ sind größer als x. Dieser Vorgang erfordert genau $l - r$ Vergleiche und $\Theta(l - r)$ Zeit. Anschließend wird rekursiv **rqsort**$(l, p-1)$ aufgerufen (falls $l < p - 1$ ist) und **rqsort**$(p + 1, r)$ (falls $p + 1 < r$ ist).

Durch einen einfachen Induktionsbeweis über die Größe $k = r - l$ zeigt man, dass **rqsort**(l, r) tatsächlich das Teilarray `A`$[l..r]$ sortiert. Die Frage ist, wie groß die erwartete Anzahl von Schlüsselvergleichen und wie groß die erwartete Laufzeit ist. Hierbei wird der Erwartungswert über die Zufallsentscheidungen des Algorithmus (bei der Wahl der Pivotelemente) gebildet, nicht etwa über verschiedene Inputs.

Wir geben den Algorithmus nochmals im Zusammenhang an:

Algorithmus 1.4.1 (RandomizedQuicksort).
Input: Ein Array `A`$[1..n]$ mit Einträgen $a_1, \ldots, a_n$, alle verschieden.

Aufruf: if $n > 1$ **then rqsort**$(1, n)$.

Dabei ist **rqsort** die folgende rekursive, randomisierte Prozedur:

(1) **Prozedur rqsort**(ℓ, r) // weiß: $1 \leq \ell < r \leq n$
(2) wähle *zufällig* ein $t \in \{\ell, \ell + 1, \ldots, r\}$ // (uniforme Verteilung)
(3) $x \leftarrow$ `A`$[t]$; // x heißt *Pivotelement* oder *partitionierendes Element*
(4) Aufruf der Prozedur **partition**(ℓ, r, t, p), mit Ausgabeparameter p:
    Transportiere die `A`$[i]$, $\ell \leq i \leq r$, mit `A`$[i] < x$
        in die Positionen `A`$[\ell]$,…,`A`$[p-1]$,
    x nach `A`$[p]$,
    und die `A`$[i]$, $\ell \leq i \leq r$, mit `A`$[i] > x$
        in die Positionen `A`$[p+1]$,…,`A`$[r]$;
(5) **if** $\ell < p - 1$ **then rqsort**$(\ell, p - 1)$;
(6) **if** $p + 1 < r$ **then rqsort**$(p + 1, r)$;

Zur Analyse beobachten wir:

- Ein Aufruf $\mathbf{rqsort}(l, r)$ kostet $r - l$ Vergleiche und Zeit $\Theta(r - l)$, wenn man die rekursiven Aufrufe nicht mitzählt.
- Es gibt insgesamt maximal $n - 1$ Aufrufe von **rqsort**. (Dies liegt daran, dass ein Eintrag, der im Aufruf $\mathbf{rqsort}(l, r)$ Pivotelement war, nicht in davon ausgelösten rekursiven Aufrufen vorkommen kann, und die Mindestlänge des Arrays in einem Aufruf 2 ist. Es kann tatsächlich $n - 1$ Aufrufe geben.)

Den gesamten Zeitaufwand erhalten wir durch Summation über alle diese Aufrufe. Die fixen Kosten aller Aufrufe liefern Zeitaufwand $O(n)$, die variablen Kosten sind

$$\sum_{\substack{1 \le l < r \le n \\ \text{Aufruf } \mathbf{rqsort}(l,r) \text{ erfolgt}}} \Theta(r - l) = \Theta\Bigg(\underbrace{\sum_{\substack{1 \le l < r \le n \\ \text{Aufruf } \mathbf{rqsort}(l,r) \text{ erfolgt}}} (r - l)}_{=: C}\Bigg)$$

Dabei gibt die Zufallsvariable C die Anzahl aller ausgeführten Vergleiche an. Die erwartete Anzahl von Vergleichen ist der Erwartungswert $\mathbf{E}(C)$ (den wir im Folgenden bestimmen werden); die erwartete Rechenzeit ist $O(n + \mathbf{E}(C))$.

Erinnerung: $b_1 < \cdots < b_n$ ist die Folge der Eingabezahlen *in sortierter Reihenfolge*. Wir definieren:

$$X_{ij} = [b_i \text{ und } b_j \text{ werden verglichen}] = \begin{cases} 1 & \text{, falls } b_i \text{ und } b_j \text{ verglichen werden,} \\ 0 & \text{, andernfalls.} \end{cases}$$

Die X_{ij} sind sogenannte ***Indikator-Zufallsvariablen***. Offensichtlich gilt

$$C = \sum_{1 \le i < j \le n} X_{ij} \text{ , also } \mathbf{E}(C) = \sum_{1 \le i \neq j \le n} \mathbf{E}(X_{ij}), \tag{1.4.13}$$

wegen der Linearität des Erwartungswertes (siehe Kapitel 2).

Da X_{ij} nur die Werte 0 und 1 annimmt, gilt

$$\mathbf{E}(X_{ij}) = \mathbf{Pr}(X_{ij} = 1) = \mathbf{Pr}(b_i \text{ und } b_j \text{ werden verglichen}),$$

was die Bestimmung des Erwartungwertes sehr vereinfacht.

Betrachte $1 \le i < j \le n$ und dazu die Menge $I_{ij} = \{b_i, b_{i+1}, \ldots, b_j\}$ der Eingabezahlen zwischen b_i und b_j. Beobachte den Ablauf des Algorithmus, wie er sich

durch die rekursiven Aufrufe hindurch entwickelt. Solange kein Element von I_{ij} als Pivotelement gewählt wird, werden bei der Partitionierung stets alle Elemente von I_{ij} im gleichen Teilarray landen. Genau in dem Moment, in dem zum ersten Mal ein Element von I_{ij} Pivotelement wird, fällt die Entscheidung darüber, ob b_i und b_j verglichen werden oder nicht. Wenn entweder b_i oder b_j dieses Pivotelement ist, erfolgt der Vergleich, wenn einer der Einträge $\{b_{i+1}, \ldots, b_{j-1}\}$ das erste ist, landen b_i und b_j bei der Partitionierung in verschiedenen Teilarrays und werden nie verglichen.

Weil Pivotelemente uniform zufällig gewählt werden, hat jedes der $j-i+1$ Elemente von I_{ij} die gleiche Wahrscheinlichkeit, das erste Pivotelement in dieser Menge zu sein. Damit gilt:

$$\mathbf{Pr}(b_i \text{ und } b_j \text{ werden verglichen}) = \frac{|\{i,j\}|}{|\{i, i+1, \ldots, j\}|} = \frac{2}{j-i+1}. \qquad (1.4.14)$$

Wenn wir die „2“ auf Paare $i < j$ und $j < i$ verteilen, erhalten wir

$$\mathbf{E}(C) = \sum_{1 \leq i \neq j \leq n} \frac{1}{j-i+1}, \qquad (1.4.15)$$

also ist $\mathbf{E}(C)$ die Summe aller Einträge in der folgenden Matrix:

$$\begin{pmatrix} 0 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{n-1} & \frac{1}{n} \\ \frac{1}{2} & 0 & \frac{1}{2} & \frac{1}{3} & \ddots & \cdots & \frac{1}{n-1} \\ \frac{1}{3} & \frac{1}{2} & 0 & \frac{1}{2} & \ddots & & \vdots \\ \vdots & \frac{1}{3} & \frac{1}{2} & 0 & \ddots & & \frac{1}{4} \\ \vdots & & \ddots & \ddots & \ddots & & \frac{1}{3} \\ \frac{1}{n-1} & & & \ddots & \ddots & \ddots & \frac{1}{2} \\ \frac{1}{n} & \frac{1}{n-1} & \cdots & \cdots & \frac{1}{3} & \frac{1}{2} & 0 \end{pmatrix}.$$

Den Wert der Summe kann man leicht direkt hinschreiben, indem man entlang der (Neben-)Diagonalen addiert und die Symmetrie beachtet. Summand $\frac{1}{k}$ kommt exakt $2(n-k+1)$-mal vor. Wir benötigen noch die *n-te harmonische Zahl*

$$H_n := 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}, \text{ für } n \geq 1.$$

Damit gilt:

$$\begin{aligned}\mathbf{E}(C) &= 2 \cdot \left(\sum_{2 \le k \le n} \frac{n-k+1}{k} \right) \\ &= 2 \cdot ((n+1)(H_n - 1) - (n-1)) \\ &= 2(n+1)H_n - 4n.\end{aligned}$$

Man weiß, dass $\ln n < H_n < 1 + \ln n$ gilt. Damit erhalten wir $\mathbf{E}(C) = 2n \ln n - O(n)$ oder $\mathbf{E}(C) = (2\ln 2) n \log_2 n - O(n)$, wobei $2 \ln 2 = 1.386\ldots$ die „Quicksort-Konstante“ ist.

Man kann sogar den linearen Term genau bestimmen: Mit dem Wissen

$$H_n = \ln n + \gamma + \frac{1}{2n} - O(1/n^2),$$

wobei $\gamma = 0.5772156649\ldots$ die „Euler-Mascheroni-Konstante“ ist, erhält man

$$\mathbf{E}(C) = (2\ln 2) n \log_2 n - (4 - 2\gamma)n + 2\ln n + O(1),$$

mit $4 - 2\gamma = 2.86\ldots$.

Bemerkung: Was ist eigentlich der Unterschied zwischen der Analyse von Quicksort mit deterministischer Pivotauswahl („immer $x = A[\lfloor (l+r)/2 \rfloor]$ wählen“) und der randomisierten Variante? Rein rechnerisch ergibt sich genau die gleiche Formel, wenn man annimmt, dass jede Anordnung der Eingabe die gleiche Wahrscheinlichkeit hat. Bei deterministischem Quicksort gibt es aber immer (worst-case-)Inputs, auf denen sich das Verfahren schlecht verhält, das heißt, Zeit $\Omega(n^2)$ benötigt. Bei der randomisierten Variante gilt dies nicht mehr: Auf jedem beliebigen Input $(a_1, \ldots, a_n)$ ist die *erwartete* Laufzeit $O(n \log n)$. Quadratische Laufzeiten treten auch hier auf, aber mit verschwindend kleiner Wahrscheinlichkeit. Die Randomisierung führt also zur Eliminierung von worst-case-Inputs.

A Illustrationen zu 1.4

Es seien $b_1 < \ldots < b_n$ die Eingabezahlen in $A[1..n]$ in aufsteigender Anordnung. In den Bildern wird angenommen, dass die Eingaben einfach $1, 2, \ldots, n$ sind (in irgendeiner Anordnung).

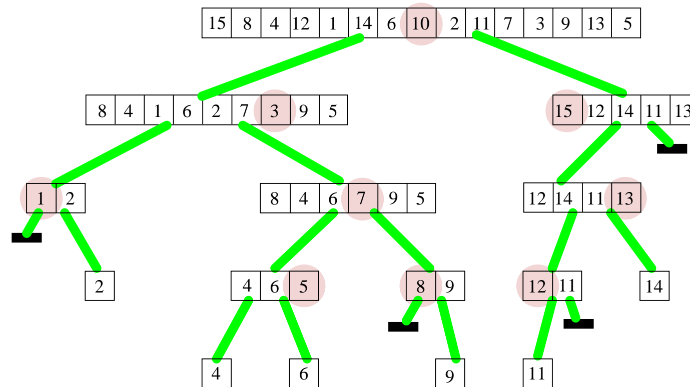

Abbildung A.0.5: Ablauf von Quicksort als Baum. Jedes Array der Länge > 1 steht für einen Aufruf von rqsort; die Kosten sind die Länge des Arrays. Pivots sind rosa.

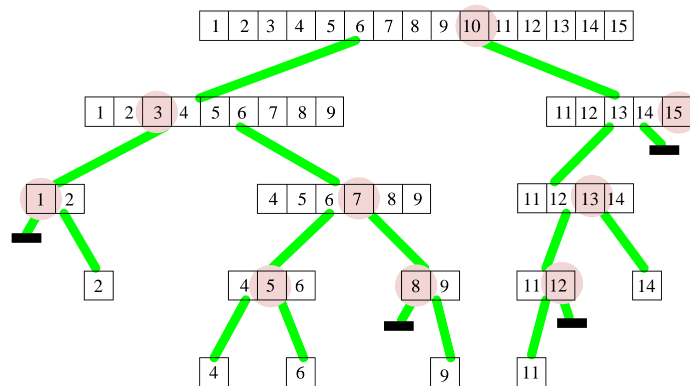

Abbildung A.0.6: Ablauf von Quicksort als Baum. Wir betrachten die ideale Situation, wo der Input aufsteigend sortiert ist (als $(b_1, \ldots, b_{15}) = (1, \ldots, 15)$). Die Wahrscheinlichkeiten für die Pivotwahlen sind exakt dieselben wie in Fig. A.0.5. Wenn der Ablauf wie in diesem Bild ist, gilt $C = 14 + 8 + 4 + 1 + 5 + 3 + 2 + 1 + 1 = 39$ und $X_{4,10} = 1$, $X_{5,11} = 0$, $X_{4,7} = 1$, $X_{3,8} = 1$, $X_{4,9} = 0$.

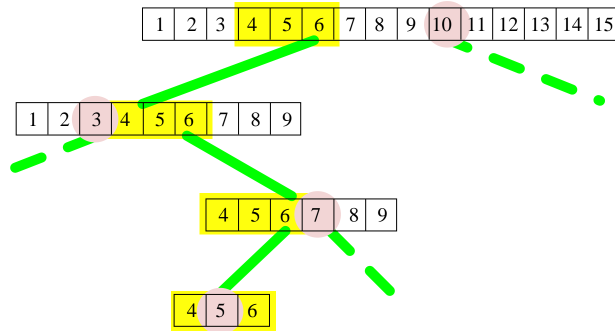

Abbildung A.0.7: Ablauf von Quicksort als Baum, idealisiert. $I_{4,6} = \{b_4, b_5, b_6\}$ (hier $= \{4, 5, 6\}$). Die Wahrscheinlichkeit, dass 4 und 6 verglichen werden, ist $\frac{2}{|\{4,5,6\}|} = \frac{2}{3}$. In diesem Beispiel (Pivot 5) passiert dies nicht, $X_{4,6} = 0$.

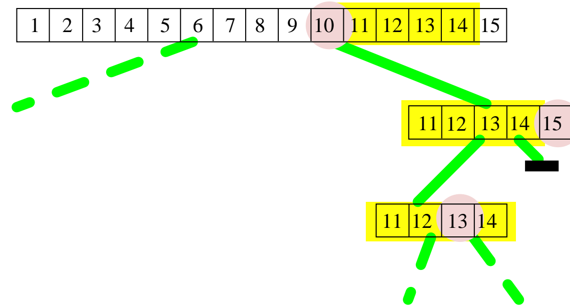

Abbildung A.0.8: Ablauf von Quicksort als Baum, idealisiert. $I_{11,14} = \{11, 12, 13, 14\}$. Die Wahrscheinlichkeit, dass 11 und 14 verglichen werden, ist $\frac{2}{|\{11,12,13,14\}|} = \frac{2}{4}$. In diesem Beispiel (Pivot 13) passiert dies nicht, $X_{11,14} = 0$.

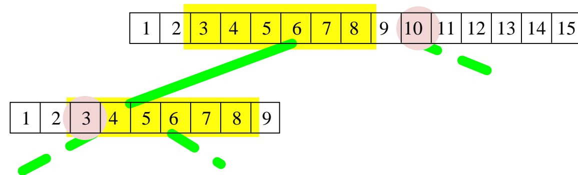

Abbildung A.0.9: Ablauf von Quicksort als Baum, idealisiert. $I_{3,8} = \{3, 4, 5, 6, 7, 8\}$. Die Wahrscheinlichkeit, dass 3 und 8 verglichen werden, ist $\frac{2}{|\{3,4,5,6,7,8\}|} = \frac{2}{6}$. In diesem Beispiel (Pivot 3) passiert dies, $X_{3,8} = 1$.

Betrachte $I_{ij} = \{b_i, \ldots, b_j\}$. Wir beobachten diese Menge, während der Algorithmus auf $A[1..n]$ abläuft. Anfangs sind alle Elemente von I_{ij} im großen Array $A[1..n]$. Nun nehmen wir an, dass $A[\ell..r]$ alle Elemente von I_{ij} enthält und **rqsort**(ℓ, r) aufgerufen wird. Wenn das Pivotelement x kleiner ist als b_i oder größer als b_j, dann wandern alle Elemente von I_{ij} in das gleiche Teilintervall, und wir beobachten weiter. Nur wenn ein Element von I_{ij} als Pivot gewählt wird, „passiert etwas".

Wenn $x = b_i$ oder $x = b_j$, werden b_i und b_j verglichen, wenn x echt zwischen b_i und b_j liegt, steckt die Partitionierungsprozedur b_i in das linke Teilarray und b_j das rechte Teilarray, also werden sie nie verglichen.

Weil alle Elemente von I_{ij} dieselbe Wahrscheinlichkeit haben, als Pivot gewählt zu werden, gilt:

$$\mathbf{Pr}(b_i, b_j \text{ werden verglichen}) = \frac{|\{b_i, b_j\}|}{|I_{ij}|} = \frac{2}{j - i + 1}.$$