Stochastische Algorithmen I.

PETER EICHELSBACHER, BOCHUM

Zusammenfassung: *Mit Hilfe der Grundbegriffe der Wahrscheinlichkeitsrechnung lassen sich sogenannte stochastische Algorithmen auch im Schulbereich untersuchen. In einer ersten Darstellung klären wir den Unterschied zu deterministischen Lösungswegen, erinnern an ein Kartenspiel und stellen den Quicksort-Algorithmus und seine randomisierte Variante vor.*

1 Einleitung

Was ist ein stochastischer Algorithmus? Sehr einfach beantwortet ist es ein Algorithmus, bei dem der Zufall mit ins Spiel kommt. Was könnte das Interesse sein, einen bekannten, nicht vom Zufall beeinflussten Algorithmus (man nennt dies häufig einen deterministischen Algorithmus) zu randomisieren?

Zwei Motivationen sollen hier besprochen werden.

1. Algorithmen sind erfunden worden für Situationen *endlicher* Probleme, etwa das Sortieren von Gegenständen nach einer Ordnung, z.B. das Sortieren einer Liste von Zahlen, das Prüfen zweier Zahlenreihen auf Gleichheit, das Auffinden eines optimalen Wertes einer Funktion auf einer endlich großen Menge. Algorithmen suchen dabei möglichst effizient alle Möglichkeiten durch, und daher muß Endlichkeit gefordert werden. Doch schon einfachste Fragestellungen der obigen Art für endliche Situationen können zu Laufzeiten der Algorithmen führen, die z.B. die Lebensdauer der Erde bei weitem übersteigt, und benötigen Speicherplatz, der auch bei modernen Rechnern nicht zur Verfügung steht. Nun versucht man sein Glück mit randomisierten Algorithmen, um die Laufzeit zu verkürzen und den Speicherplatzbedarf zu verkleinern.

2. Algorithmen sollen bei allen denkbaren Eingaben korrekt und schnell arbeiten. Häufig gibt es aber Eingabewerte, die zu einer sehr ungünstigen Laufzeit führen. Die Laufzeit-Analyse eines Algorithmus muß aber solche *worst case* Eingaben berücksichtigen, auch wenn viele Eingaben zu einem guten Verhalten des Algorithmus führen. Beim *auswürfeln* eines Eingabewertes würde man nun erwarten, dass sich der Algorithmus mit hoher Wahrscheinlichkeit gut verhält. Der stochastische Algorithmus sollte nun für alle Eingaben eine Laufzeit liefern, die mit hoher Wahrscheinlichkeit gutartig ist. Ungünstige Verläufe mögen mit geringer Wahrscheinlichkeit eintreten. Liegt ein Algorithmus vor, der eine sehr große Menge möglicher Fälle zu untersuchen hat, liegt die Hoffnung im randomisierten Algorithmus darin, dass das Erkunden eines kleinen Teils der großen Menge mit großer Wahrscheinlichkeit schon eine relevante Information erbringt.

Stochastische Algorithmen werden also in der Regel entweder die Lösung nicht immer mit Sicherheit finden oder nicht mit absoluter Präzision.

Wir wollen ein Beispiel geben, um das Wort *Randomisierung* zu entdecken: Ein verrückter Millionär sendet an 20 Personen einen Brief mit dem Text: *diesen Text haben weitere 19 Personen erhalten. Wählen Sie aus, ob Sie mir zurückschreiben oder nicht. Kommt bei mir genau ein Brief an, so erhält der Absender eine Million Euro. Falls keiner oder mehr als einer eintrifft, so erhält niemand etwas. Der Kontakt untereinander ist verboten*. Was ist die beste Strategie? Wenn es eine beste Strategie gibt, so kann man davon ausgehen, dass jeder der 19 Mitspieler diese Strategie unabhängig von den anderen verwendet. Es gibt aber nur die Strategie, einen Brief abzuschicken oder nicht. Beide Strategien haben aber nach den genannten Spielregeln keinen Erfolg! Was passiert nun bei einer Randomisierung? Wir schicken mit Wahrscheinlichkeit p mit $p \in (0,1)$ einen Brief ab. Dann gewinnt man mit Wahrscheinlichkeit $p(1-p)^{19}$ eine Million. Der Wert dieser Funktion in p ist maximal bei $p_0 = \frac{1}{20}$, wie man leicht sieht: Sei $f(p) = p(1-p)^{19}$, dann ist $f'(p) = (1-p)^{19} - 19p(1-p)^{18}$. Es gilt $f'(p) = 0$ genau dann wenn $(1-p)^{19} = 19p(1-p)^{18}$, also $1 = 20p$. Daraus ergibt sich p_0. Wenn wir uns eine Zufallsmaschine bauen, die mit Wahrscheinlichkeit $\frac{1}{20}$ entscheidet, so schicken wir im positiven Fall den Brief ab, sonst nicht. Der randomisierte Algorithmus ist jedem deterministischen Algorithmus überlegen. Zugegeben, dieses Beispiel ist etwas konstruiert, aber es hebt auf einfache Weise mögliche Vorteile eines randomisierten Algorithmus gegenüber einem deterministischen hervor. In diesem Artikel wollen wir uns mit dem Sortieren einer langen Liste von Zahlen befassen. Hier ist eine randomisierte Version eines Sortier-Algorithmus unser Focus.

2 Erinnerung an ein Kartenspiel: Patience

Erinnerung soll hierbei bedeuten, dass wir das folgende Patience-Spiel in der Arbeit (2) bereits einmal vorgestellt haben: Wir benötigen ein Kartenspiel von N Karten, die "linear geordnet" sind, d.h. für je zwei Karten lässt sich entscheiden, welche die höher wertige ist. Beispielsweise kann man von einem Kartenspiel nur die Herzkarten entnehmen und diese mit der Ordnung

$$As < 2 < 3 < \cdots < B < D < K$$

versehen. Die Karten werden gemischt und auf einen Stapel gelegt. Von diesem Stapel wird nun sukzessive eine nach der anderen Karte von oben entnommen und in einem von mehreren Stapeln abgelegt. Die Regel dabei ist, dass eine Karte *nicht* auf einen Stapel gelegt werden kann, wenn sie höher als dessen oberste Karte ist. Gibt es keinen Stapel, auf den eine neu gezogene Karte gelegt werden kann, muss ein neuer Stapel angefangen werden. Ziel ist es, das Spiel mit so wenig Stapeln wie möglich zu beenden.

Wir wollen nun annehmen, das Kartenspiel sei gut gemischt (wieviele Mischvorgänge dafür notwendig sind, steht auf einem anderen Blatt und kann hier nicht im Detail erläutert werden).

Wir probieren nun, unser einfaches Patience-Spiel zu spielen. Wie kann man erreichen, dass zum Schluss so wenig Stapel wie möglich ausgelegt sind? Eine Strategie liegt auf der Hand: Wir legen jede neue Karte auf den erstmöglichen Stapel. Diese Strategie wollen wir die *gierige Strategie* nennen. Schauen wir sie uns anhand eines Beispiels an:

Beispiel 2.1 *Der gemischte Kartenstapel von $N = 10$ Karten sei gegeben durch*

$$5 \quad 3 \quad 6 \quad 7 \quad 4 \quad 9 \quad 10 \quad 1 \quad 8 \quad 2$$

Spielen wir damit Patience gemäß der gierigen Strategie, erhalten wir die folgende Abfolge von Stapeln:

```
5    3    3      3     3 4
     5    5 6    5 6 7 5 6 7
```

```
                          1
3 4        3 4            3 4
5 6 7 9    5 6 7 9 10     5 6 7 9 10
```

```
1              1 2
3 4   8        3 4   8
5 6 7 9 10     5 6 7 9 10
```

Wir haben unser Spiel also mit fünf Stapeln beendet. Aber wie gut ist das? In der Arbeit (2) konnten wir herleiten, dass in dem oben dargestellten Patience-Spiel die gierige Strategie optimal ist (und mehr). Die geschilderte Variante des Patience-Spiels wird auch *Patience-Sortieren* genannt, und zwar aus folgendem Grund: Am Ende des Spiels liegt Karte 1 oben im ersten Stapel, wenn man Karte 1 wegnimmt, liegt Karte 2 oben auf einem der Stapel, wenn man diese wegnimmt, liegt Karte 3 oben auf einem der Stapel usw.. Mit Hilfe dieses Vorgehens kann man also die Karten von Hand sortieren. Wir haben also bereits einen spielerischen Sortiert-Algorithmus gefunden. Wir hatten weiter in (2) nach einer möglichen Computersimulation des obigen Patience-Spiels gefragt. Dieser muss ein mathematisches Modell zugrunde liegen. Hierbei ist die erste Frage, die es zu klären gilt: Was bedeutet es, dass wir das Kartenspiel zu Beginn gut mischen?

Der Sinn eines solchen Mischens ist es offenbar, die Karten aus einer möglicherweise bekannten Anfangssituation so durcheinander zu bringen, dass die Reihenfolge nicht mehr vorhersagbar ist. In mathematischer Sprechweise erhalten wir durch Mischen eine Permutation unserer Karten, oder, wenn wir die Karten mit 1 bis N durchnummerieren, eine Permutation der Zahlen $1,\ldots,N$. Die Menge aller Permutationen von $\{1,\ldots,N\}$ heißt *symmetrische Gruppe* und wird mit S_N bezeichnet.

Ein Kartenspiel können wir als gut gemischt bezeichnen, wenn wir nach dem Mischen keine Idee mehr haben, welche Permutation vorliegt, d.h. wenn alle Permutationen $\sigma \in S_N$ gleich wahrscheinlich sind. Nun hat S_N $N!$ viele Elemente, d.h. wir dürfen das Kartenspiel als gut gemischt bezeichnen, wenn für die Situation nach dem Mischen gilt:

$$P(\sigma \text{ liegt vor }) = P(\sigma) = \frac{1}{N!}.$$

Wie lässt sich nun am praktischsten eine Permutation $\sigma \in S_N$ simulieren? Formal lässt sich eine Permutation σ als eine Folge von N Paaren $(i,\sigma(i))$ aufschreiben mit $1 \leq i \leq N$ und $1 \leq \sigma(i) \leq N$. Hierbei bezeichnet $\sigma(i)$ den Platz an dem die Ziffer i in der Permutation σ erscheint. Die Permutation

$$\sigma = \{(1,2),(2,1),(3,3)\} \in S_3$$

bedeutet, dass Karte 2 nach dem Mischen auf Platz 1, Karte 1 auf Platz 2 und Karte 3 auf Platz 3 gelandet ist. Manchmal schreibt man dafür kürzer $\sigma = (2\ 1\ 3) \in S_3$. Nun ist es offenbar gar nicht wichtig, dass die $\sigma(i) \in \{1,\ldots,N\}$ sind. Wichtig ist allein, dass für $i \neq j$ auch $\sigma(i) \neq \sigma(j)$ ist. Dann kann man die $i \in \{1,\ldots,N\}$ in der auf- oder absteigenden Reihenfolge der $\sigma(i)$ aufschreiben und erhält eine zufällige Permutation. Dies ist beispielsweise gewährleistet, wenn man für jedes i eine Zufallsvariable $\sigma(i)$ zieht, die unabhängig von allen anderen und uniform im Intervall $[0,1]$ verteilt ist (eine solche ist leicht auf einem Computer zu erzeugen). Hierbei bemerke man, dass für zwei uniform auf $[0,1]$ verteilte Zufallsvariablen X,Y mit Wahrscheinlichkeit eins $X \neq Y$ gilt. Es wird also stets für $i \neq j$ entweder $\sigma(i) < \sigma(j)$ oder umgekehrt $\sigma(j) < \sigma(i)$ gelten. Ein Rezept zum Erzeugen einer Zufallspermutation lautet genauer also so:

- Erzeuge unabhängige, uniform in $[0,1]$ verteilte Zufallszahlen $\sigma(1),...,\sigma(N)$.
- Ordne $\{1,\ldots,N\}$ so, dass
  $$\sigma(i_1) < \sigma(i_2) < .. < \sigma(i_N)$$
  gilt. Hierbei ist $\{i_1,\ldots,i_N\}$ die neue Anordnung der Menge $\{1,\ldots,N\}$.
- Dann ist $(i_1,i_2,...,i_N) \in S_N$ eine zufällige Permutation.

Es bleibt also festzuhalten: das Patiencespiel bietet eine Möglichkeit, die Karten zu sortieren. Wenn Sie sich einen größeren Stapel von bereits korregierten Klausuren vornehmen und ihn alphabetisch sortieren wollen, so spielen Sie doch einfach Patience damit, es erfolgt eine (schnelle ?) alphabetische Sortierung. Interessant ist, dass man herausgefunden hat, dass man diesen Sortier-Algorithmus implementieren kann mit $N\log N - N\log\log N + O(N)$ Vergleichen (zweier Werte) maximal. Desweiteren hat kein anderer Algorithmus ein besseres *worst case* Verhalten. Dies findet man in (3). Es bezeichnet hier $O(N)$ eine Größe, die durch ein cN mit c konstant nach oben abgeschätzt werden kann.

Die Simulation einer Mischung des Kartendecks kann am Rechner mit Hilfe der uniform verteilten Zufallszahlen erfolgen, allerdings benötigt man hier intern bereits einen Sortieralgorithmus!

3 Quicksort

Der Algorithmus mit Hilfe des Patience-Spiels ist natürlich nicht eine Vorgehensweise, auf die man in natürlicher Art und Weise stoßen würde. Nach der spielerischen Variante, die ein Sortieren ermöglicht, und deren Computer-Implementierung selbst einen Sortier-Algorithmus benötigt, stellen wir nun den gebräuchlichsten Sortier-Algorithmus vor, den sogenannten Quicksort-Algorithmus. Auf ihn kann man kommen, seine Analyse soll hier mittels der Methoden der Wahrscheinlichkeitsrechnung erfolgen. Die Laufzeit-Analyse einer randomisierten Form wird dabei recht einfach erfolgen können.

Gegeben sei eine Menge S von n paarweise verschiedenen Zahlen. Sie soll der Größe nach sortiert werden. Zunächst wählt man ein Element y aus. Man kann zum Beispiel immer das erste Element nehmen, man kann es aber auch zufällig auswählen. Das ausgewählte Element y nennt man *Pivot-Element*. Man vergleicht dieses Pivot-Element nun mit allen anderen Zahlen und partitioniert die Zahlen in zwei Mengen; die eine umfaßt alle Zahlen, die kleiner sind als y (genannt S_1), die andere alle Zahlen die größer als y sind (genannt S_2). Man benötigt dazu $n-1$ Vergleiche. Man gebe S_1, y und S_2 in dieser Reihenfolge aus. Nun wende man den beschiebenen Algorithmus (Auswahl des Pivot-Elements, Partitionierung, Ausgabe) rekursiv auf S_1 und S_2 an (zumindest solange die Mengen mehr als ein Element besitzen). Wählt man y fest (zum Beispiel immer das erste Element), sprechen wir vom *deterministischen Quicksort-Algorithmus*, wählen wir y zufällig nach der Geleichverteilung auf S, so sprechen wir vom *randomisierten Quicksort-Algorithmus*.

Das Maß für die sogenannte *Laufzeit* des Algorithmus ist nun die Gesamtanzahl der benötigten, der durchgeführten Vergleiche von zwei Zahlen. X_n bezeichne im folgenden die Anzahl der von Quicksort durchgeführten Vergleiche. Dies ist im Fall des randomisierten Quicksort-Algorithmus eine Zufallsvariable, daher die Bezeichnung X_n.

Wir wollen zunächst diskutieren, in welchen Situationen der Algorithmus besonders schnell bzw. besonders langsam den Sortier-Vorgang erledigt. Man wähle zum Beispiel als Pivot-Element immer das erste Element. Jetzt ist die Laufzeit von Quicksort im schlimmsten Fall von der Größenordnung n^2, wenn die vorliegende Liste von Zahlen bereits sortiert ist. Denn dann benötigen wir im ersten Schritt $n-1$ vergleiche, im zweiten Schritt $n-2$ Vergleiche und im letzten Schritt noch einen Vergleich, also insgesamt

$$(n-1)+(n-2)+(n-3)+\cdots+1 = n(n-1)/2$$

Vergleiche. Dieser *worst case* kann auch beim randomisierten Quicksort auftreten, nämlich genau dann, wenn der Zufall stets als Pivot-Element das kleinste oder das größte Element auswählt. Auf der anderen Seite ist Quicksort besonders schnell, falls stets der mittlere Wert, der *Median*, der zu sortierenden Mengen gewählt wird (deterministisch oder zufällig). Nur kommt hier schon die Schäche des deterministischen Version zum Vorschein: soll das Pivot-Element der Median sein, so müssen wir das mittlere Element finden. Es ist das mittlere Element einer sortierten Sequenz von Elementen. Es gibt zwar einen randomisierten Algorithmus zur Bestimmung des Medians, der weniger Vergleiche benötigt als jeder theoretisch mögliche determinsitische Algorithmus (und dieser kann an anderer Stelle einmal besprochen werden), aber die Laufzeit (die bei cn Vergleichen mit einer Konstanten c liegt) müsste bei jeder Auswahl des Pivot-Elements hinzugerechnet werden. Wenn nun aber -rein theoretisch- die randomisierte Variante von Quicksort immer als Pivot-Element den Median liefert, so sind die beiden Mengen S_1 und S_2 jeweils etwa gleich groß. Dann benötigt man jeweils etwa n Vergleiche, aber die Anzahl der Iterationen (hier Halbierungen) ist etwa $\log_2(n)$. Dies ist einfach einzusehen: $\log_2(n)$ ist diejenige Zahl, mit der man 2 potenzieren muß um n zu erhalten. Insgesamt ist also ein günstiger Verlauf des Quicksort-Algorithmus durch $X_n = n\log_2 n$ gegeben. Wir fassen zusammen: Die Anzahl der Vergleiche beim Quicksort-Algorithmus liegt zwischen $n\log_2 n$ und $n^2/2$.

Für die deterministische Variante von Quicksort kann man sich nun aufmachen und die *durchschnittliche Laufzeit* ausrechnen. Wir betrachten dazu den Ansatz, führen die Rechung aber nicht durch. Ist das Pivot-Elemenet das i-t größte Element der Liste, so haben die beiden rekursiv zu sortierenden Teilfelder S_1 und S_2 die Größe $i-1$ und $n-i$. Dann beschreibt

$$X_n = n-1+\frac{1}{n}\sum_{i=1}^{n}\left(X_{i-1}+X_{n-i}\right)$$

für $n \geq 1$ und bei Setzung $X_0 := 0$ das durchschnittliche Verhalten des Quicksorts. Man sieht:

$$X_n = n-1+\frac{2}{n}\sum_{i=1}^{n}X_{i-1}.$$

Eine solche Rekursionsgleichung kann man lösen. Eine verbreitete Methode ist die der erzeugenden Funktion. Wir lassen dies hier weg und teilen mit:

$$X_n = 2(n+1)\sum_{i=1}^{n}\frac{1}{i}-2n-4.$$

Wenn wir nun verwenden, dass $\frac{1}{j} \leq \frac{1}{x}$ für $x \in [j-1, j]$ und daher

$$1+\sum_{j=2}^{n}\frac{1}{j} \leq 1+\int_{1}^{n}\frac{1}{x}\,dx = 1+\log n,$$

so sehen wir, dass der Quicksort-Algorithmus in seiner determinstischen Form zum Sortieren von n Zahlen im Duchschnitt $n\log n + O(n)$ Vergleiche benötigt. Wieder meint $O(n)$ einen Ausdruck, der durch cn mit c konstant nach oben abgeschätzt werden kann. Der Algorithmus ist also im Mittel wesentich schneller als im worst case (n^2). Man bedenke aber, dass zum Beispiel bei Eingabe-Listen von überwiegend vorsortierten Daten die Aussage über die durchschnittliche Laufzeit wenig aussagekräftig ist! Im randomisierten Fall wird man nun aber das folgende Resultat beobachten können: *wählt man das Pivot-Element nicht deterministisch sondern zufällig, so ist bei allen(!) Eingaben die Laufzeit mit hoher Wahrscheinlichkeit gutartig.*

Zur Klärung dieser Aussage machen wir uns nun auf, im randomisierten Fall den Erwartungswert von X_n zu bestimmen.

Theorem 3.1 *Beim randomisierten Quicksort-Algorithmus beträgt die mittlere Laufzeit*

$$E(X_n) = 2(n+1)\big(1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}\big)-4n,$$

also

$$E(X_n) = 2n\log n + O(n).$$

Bevor wir den Beweis geben, hier eine schnelle Vergleichsdiskussion: da $2\log n$ ungefähr $1.39\log_2 n$ ist, muss man also im Mittel nur 39 Prozent mehr Vergeliche vornehmen als im günstigen Fall!

Beweis. Eine der fundamentalen Ideen in der Stochastik zur Bestimmung des Erwartungswertes einer Zufallsvariablen ist die Zerlegung der Zufallsvariable in eine Summe einfacherer Zufallsvariablen. Der Erwartungswert ist dann die Summe der Erwartungswerte der einfacheren Zufallsvariablen (Linearität des Erwartungswertes). Für die Menge S schreiben wir nun

$$S = \{s_{(1)}, s_{(2)}, \ldots, s_{(n)}\},$$

wobei dies die korrekte Ordnung von S sei, also $s_{(i)} < s_{(j)}$ für $i < j$. Somit bezeichnet $s_{(i)}$ das i-t kleinste Element. Es sei 1_A die Indikatorfunktion auf der Menge A, also die Funktion, die einem x den Wert 1 zuordnet, wenn $x \in A$, und sonst den Wert 0 zuordnet. Dann gilt

$$X_n = \sum_{1 \le i < j \le n} 1_{A_{ij}},$$

wobei A_{ij} das Ereignis bezeichnet, dass es im Verlauf des Algorithmus zum Vergleich von $s_{(i)}$ und $s_{(j)}$ kommt. Dann ist

$$E(X_n) = \sum_{1 \le i < j \le n} E(1_{A_{ij}}).$$

Der Erwartungswert eines Indikators 1_A ist $E(1_A) = 1P(A) + 0P(A^c) = P(A)$. Also müssen wir $P(A_{ij})$ berechnen. Wir behaupten:

$$P(A_{ij}) = \frac{2}{j-i+1}$$

für alle $1 \le i < j \le n$. Dies sieht man so: das Ereignis A_{ij} tritt genau dann ein, wenn entweder $s_{(i)}$ oder $s_{(j)}$ das erste der Elemente $s_{(i)}, \dots, s_{(j)}$ ist, das als Pivot-Element y herangezogen wird. Denn falls ein anderes Element dieser Teilliste gewählt würde - nennen wir es $s_{(k)}$ mit $i < k < j$- , dann würden $s_{(i)}$ und $s_{(j)}$ zwar je mit $s_{(k)}$ verglichen werden, aber sie würden nicht miteinander verglichen werden, denn sie gehörten dann zu verschiedenen Teilisten S_1 und S_2, die beim rekursiven Verfahren ja im Anschluss getrennt behandelt werden. Nun hat jedes der Elemente $s_{(i)}, \dots, s_{(j)}$ dieselbe Wahrscheinlichkeit, als erstes als Pivot-Element gewählt zu werden. Also folgt

$$P(A_{ij}) = \frac{2}{j-i+1}.$$

Wir müssen also - mit etwas Geduld -

$$2 \sum_{j=2}^{n} \sum_{i=1}^{j-1} \frac{1}{j-i+1}$$

ausrechnen bzw. umformen. Für jedes j ist die innere Summe $\frac{1}{j} + \frac{1}{j-1} + \cdots + \frac{1}{2}$, hat also die Darstellung $\sum_{i=2}^{j} \frac{1}{i}$. Wir erhalten

$$E(X_n) = 2 \sum_{j=2}^{n} \sum_{i=2}^{j} \frac{1}{i}.$$

Nun vertauschen wir die Summationsreichenfolge:

$$E(X_n) = 2 \sum_{i=2}^{n} \sum_{j=i}^{n} \frac{1}{i}.$$

Dies sieht man mit Hilfe eines einfachen Bildes ein:

Doppelsummen-Schema				
j \ i	2	3	⋯	n
2	*			
3	*	*		
⋮				
n	*	*	…	*

Nun ist aber die innere Summe $(n-i+1)\frac{1}{i} = (n+1)\frac{1}{i} - 1$. Damit erhalten wir

$$E(X_n) = 2(n+1) \sum_{i=2}^{n} \frac{1}{i} - 2(n-1).$$

Und nun ist die rechte Seite gleich

$$2(n+1) \sum_{i=1}^{n} \frac{1}{i} - 2(n+1) - 2(n-1) = 2(n+1) \sum_{i=1}^{n} -4n.$$

Die Asymptotik folgt nun wieder aus

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \log n + O(1).$$

Damit ist der Satz bewiesen. ■

Die mittelere Laufzeit ist etwas besser als die angedeutete im deterministischen Fall. Eine noch bessere Aussage über die Qualität des randomisierten Quicksorts ist, dass X_n mit hoher Wahrscheinlichkeit in der Nähe seines Erwartungswertes liegt. Eine solche Aussage wurde im Jahre 1991 von Rösler (7) bewiesen und liesst sich so:

Für beliebige $\lambda, \varepsilon > 0$ existiert eine Konstante $c(\lambda, \varepsilon) > 0$, so dass

$$P\big(X_n > (1+\varepsilon)E(X_n)\big) \le c(\lambda, \varepsilon)\, n^{-2\lambda\varepsilon},$$

wobei n eine natürliche Zahl ist. Dies ist die Rechtfertigung, dass für alle(!) Eingaben die Laufzeit mit hoher Wahrscheinlichkeit gutartig ist.

Nimmt man an, dass alle möglichen Eingaben, also alle Permutationen der Eingabe, gleich wahrscheinlich sind und interessiert sich dann für den Erwartungswert der Laufzeit, so kann man mittels einer leichten Umformulierung des obigen Beweises zeigen, dass auch hier die durchschnittliche Laufzeit in der Größenordnung $n \log n$ liegt. Wir führen dies nicht aus.

Es bleibt abschließend darauf hinzuweisen, dass es viele Untersuchungen zu Sortieralgorithmen gibt. Hoare hat 1961 den hier diskutierten Quicksort eingeführt, siehe (4). Andere Sortier Algorithmen tragen den Namen *Heapsort* oder *Mergesort* und werden unter anderem in (6) diskutiert. Ein gut ausgearbeitetes Skriptum ist (5).

Literatur

[1] D. Aldous, P. Diaconis, *Longest increasing subsequences: from patience sorting to the Baik-Deift-Johansson theorem;* Bull. Amer. Math. Soc. (N.S.) 36 (1999), 413–432.

[2] P. Eichelsbacher, M. Löwe, *Geduld und Zufall*; Stochastik in der Schule, Band 23 , Heft 2 (2003), 2–6.

[3] M.L. Fredman, *On computing the lenght of the longest increasing subsequence*; Discrete Mathematics, Vol. 11, (1975), 29–35.

[4] C.A.R. Hoare, *Quicksort*, Computer Journal, Vol.5, (1961), 10–15.

[5] W. König, *Stochastische Algorithmen*, Universität Leipzig (2004).

[6] U. Krengel, *Einführung in die Wahrscheinlichkeitstheorie und Statistik*, Vieweg Verlag, 5. Auflage (2000).

[7] U. Rösler, *A limit theorem for Quicksort*, Theor. Inf. Appl., Vol 25, (1991), 85–100.

Anschrift des Verfassers:

Peter Eichelsbacher
Fakultät für Mathematik
Ruhr-Universität Bochum
NA 3 / 68
44780 Bochum
peter.eichelsbacher@ruhr-uni-bochum.de