Randomisierte Algorithmen

Mentorierte Arbeit

Sabrina Wiedersheim

Betreuung:
Prof. Dr. J. Hromkovic

Zürich, 13. Juli 2008

Vorwort

In dieser mentorierten Arbeit sollen die Schüler in das Themengebiet der randomisierten Algorithmen eingeführt werden und anhand von Beispielen die Stärke des Zufalls für Algorithmen entdecken. Dabei werden vor allem zwei Typen von randomisierten Algorithmen erarbeitet: Die Methode des Überlistens von Gegnern und die Methode der Fingerabdrücke (Grundlage: Entwurfsmethoden von zufallsgesteuerten Systemen, Prof. Dr. Juraj Hromkovic). Die erste Methode wird über ein Codeknackerproblem und über das Themengebiet der Sortieralgorithmen (Quicksortalgorithmus) erklärt. Der zweite Teil baut auf dem Problem der Verifikation von Matrixmultiplikationen auf.
Ziel der Arbeit ist auch vor allem die Anwendung der Wahrscheinlichkeitstheorie in der Informatik. Von den Schülern werden folgende Kenntnisse vorausgesetzt:

- Gute Grundlage in der Wahrscheinlickeitstheorie und Kombinatorik (Skript: Einführung in die Wahrscheinlichkeitstheorie, Prof. Dr. Juraj Hromkovic) und Kentnisse über den Erwartungswert (Linearität des Erwartungswertes)

- Gewisse erste Erfahrungen in der Informatik: Was ist ein Algorithmus? Notation von Algorithmen (if- und for-Schleifen, input, output, ...) und erste Erfahrung von Laufzeitanalysen ($\mathrm{O}(n)$, $\mathrm{O}(log(n))$, $\mathrm{O}(n^2)$, ...) sind von Vorteil, können aber allenfalls auch umgangen werden.

- Analysis: Die Notation von Summen über das Summenzeichen sollte den Schülern bekannt sein. Die Kenntis der Harmonischen Zahl H_n wäre von Vorteil.

- Die Themengebiete der Sortieralgorithmen und der Matrizen werden in dieser Arbeit zwar eingeführt, aber nur sehr knapp. Es ist deshalb von Vorteil, wenn die Schüler dabei schon gewisse Kentnisse mitbringen oder wenn sie bei diesen Einführungsabschnitten zusätzliche Unterstützungen erhalten.

Inhaltsverzeichnis

1 Einführung

Was ist ein Algorithmus? Ein Algorithmus ist ein Verfahren, das uns (oder dem Computer) Schritt für Schritt Anweisungen gibt, was zu tun ist. Solche Anleitungen gibt es nicht nur in der Mathematik und Informatik, sondern sie begegnen uns auch stetig im alltäglichen Leben. Sei es die Schritt für Schritt Installationsanleitung des neuen Computerspiels, das Backrezept für Grossmutters Lieblingtorte oder die Bauanleitung für Vaters Büroregal - überall sind wir mit Anweisungen konfrontiert, die uns Schritt für Schritt das Vorgehen für das Erreichen eines gewissen Ziels erläutern. Je nach dem an wen sich diese Anleitungen richten, müssen sie unterschiedlich ausführlich formuliert werden!

Was passiert, wenn man bei solchen Verfahren gewisse Entscheidungen und Schritte zufällig wählt? - Wenn man nicht mehr das ganze Vorgehen von Beginn bis Ende exakt vorschreibt, sondern zwischendurch den Zufall walten lässt? Beim Kuchenrezept wäre wahrscheinlich der Einbau einer zufälligen Wahl zwischen 100g Zucker oder 100g Salz keine vorteilhafte Idee. Wir werden aber in den nächsten drei Kapiteln andere, auch aus unserem Alltag gegriffene, Probleme bertrachten in denen der Einbau des Zufalls in den Lösungsverfahren Vorteile bringen wird! Wir werden sehen, dass man sich für einen Gegner undurchschaubar machen kann, in dem man Lösungsstrategien verwendet, die gewisse Entscheidungen dem Zufall überlassen. So werden wir einem Jungen helfen den Zahlencode seiner Schwester schneller zu knacken. Und wir werden diese Idee des Überlistens von einem Gegner auch für das schnellere Sortieren von Zahlen anwenden können. Dann werden wir aber auch sehen, dass man mit dieser Idee der zufälligen Entscheidungen nicht nur einen Gegner irritieren kann, sondern, dass man damit auch schneller grosse Rechnungen überprüfen kann, wenn man eine kleine Fehlerwahrscheinlichkeit zulässt.

In der Informatik unterscheidet man deshalb zwei grosse Klassen von Algorithmen: Jene Klasse von Algorithmen, die ganz ohne zufällige Entscheidungen funktionieren, und eine zweite Klasse von Algorithmen, die teilweise vom Zufall gesteuert werden. In der Fachsprache spricht man auch von *deterministischen Algorithmen* und *randomisierten Algorithmen* (zufälligen Algorithmen).

2 Codeknacker

In diesem Abschnitt werden wir ein erstes Beispiel eines randomisierten Verfahrens und seine Vorteile sehen. Wir betrachten dazu folgende Situation: Anna schreibt ihr Tagebuch auf dem Computer und ihr kleiner Bruder Tim möchte sie ausspionieren. Da sich Anna aber dessen bewusst ist, schützt sie ihr Dokument mit einem 4-stelligen Zahlenpasswort.

Aufgabe 1.

(a) Wie viele mögliche solche 4-stellige Passwörter mit den Ziffern $\{0, 1, 2, \ldots, 9\}$ *stehen Anna zur Auswahl?*

(b) Angenommen die Wahrscheinlichkeit für die Wahl ihres Passwortes sei für jedes mögliche Passwort gleich gross. Mit welcher Wahrscheinlichkeit findet Tim ihr Passwort, wenn er nur drei Versuche hat?

Nun ist aber der Passwortschutz von Annas Worddokument nicht ganz so gut, wie jener eines Bankautomaten. Bei ihrem Passwort gibt es keine automatische Sperre nach einer gewissen Anzahl Versuche. Tim kann also beliebig viele Passwörter ausprobieren. Wie muss er vorgehen um auf sicher Annas Code zu knacken?
Da er keine Präferenzen von Annas Passwörtern kennt, bleibt ihm nichts anderes übrig, als ein Passwort nach dem anderen auszuprobieren. Und damit er kein Passwort aus Versehen vergisst, muss er sich für eine bestimmte Reihenfolge entscheiden, wie zum Beispiel: $0000, 0001, 0002, 0003, \ldots$,9999.

Aufgabe 2. *Angenommen die Passwortwahl von Anna sei, wie bereits in Aufgabe 1(b), zufällig. Tim hat genügend Zeit und ist sehr geduldig: Am Mittwoch Nachmittag sitzt er 4 Stunden vor Annas Computer und gibt Passwörter ein. Pro Minute kann er 20 Passwörter testen. Mit welcher Wahrscheinlichkeit wird er Annas Code bis in 4 Stunden geknackt haben?*

Nun kennt aber Anna ihren kleinen Bruder gut. Sie weiss, wie Tim vorgehen wird um ihr Passwort herauszufinden. So wählt sie ihr Passwort nicht mehr zufällig, sondern sie gibt gerade das letzte Passwort aus Tims Reihenfolge ein. Der nichts ahnende Tim hat keine Chance mehr. Er wird vergeben 4 Stunden vor dem Computer sitzten. Denn die Wahrscheinlichkeit, dass er das Passwort knacken wird, ist in diesem Fall 0 (er kann in den 4 Stunden nicht alle Codes testen und kommt somit niemals zum letzten Passwort seiner Reihenfolge). Können wir Tim helfen? Was ist sein Problem?
Tim hat eine recht eigeschränkte Vorgehensweise. Bevor er mit dem Testen beginnt, weiss er genau in welcher Reihenfolge er die Passwörter durchgehen

wird und ermöglicht somit Anna (Gegner) die Aufgabe schwerst möglich zu stellen. Sein Verfahren ist deterministisch (vor dem Start des Verfahrens ist klar wie es ablaufen wird). Tims möglicher Ausweg ist nun der Einbau vom Zufall: Jedes mal, wenn er hinter Annas Passwort geht, soll er sich zufällig eine neue Reihenfolge aller möglichen Reihenfolgen zusammenstellen und die Passwörter so testen.

Aufgabe 3.

(a) Aus wievielen möglichen Passwortreihenfolgen kann Tim zufällig eine auswählen?

(b) Mit welcher Wahrscheinlichkeit ist dann ein beliebiges Passwort, wie z.B. 2593, das erste Passwort in seiner Reihenfolge? Und mit welcher Wahrscheinlichkeit ist es das letzte?

(c) Tim arbeitet wieder 4 Stunden an Annas Computer (20 Passwörter pro Minute). Anna wählte das Passwort 9999. Wie gross ist die Wahrscheinlichkeit, dass Tim ihr Passwort findet? Wie gross ist sie, wenn Anna 0000 gewählt hätte?

Mit diesem Einbau des Zufalls, kann Tim den Spiess wieder umdrehen. Anna kann sein Vorgehen nicht mehr durchschauen. Die Wahrscheinlichkeit, dass Tim ihr Passwort in den 4 Stunden findet, ist wieder gleich gross wie, in dem Fall, als Anna ihren Code zufällig wählte und Tim das Passwort nach einer bestimmten Reihenfolge testete. Diese Methode der Randomisierung, nennt man auch „Überlisten des Gegners“. Bei einem deterministischen Lösungsverfahren / Algorithmus kann einem ein „Gegner“ das Problem schwerst möglich machen. Nach der Randomisierung kann der Gegner das Vorgehen aber nicht mehr durchschauen! Im nächsten Kapitel werden wir ein weiteres solches Beispiel antreffen.

3 Quicksort

3.1 Sortieralgorithmen

Das Sortieren von Elementen kommt in unserem Alltag immer wieder vor: Das reicht vom Handkarten sortieren beim Jassen am Spielnachmittag bis zum Quittungen und Rechnungen sortieren für die Buchhaltung. Auch in der Informatik spielt das Sortieren von Datensätzen eine grosse Rolle. Es ermöglicht einem den Überblick über eine grosse Menge von Daten und auch das schnelle Wiederfinden von Daten. Was würde uns ein Telefonbuch nützen, wenn diese Nummern nicht nach Ort und Namen sortiert wären? Für praktische Anwendungen ist es natürlich wichtig, dass man Verfahren hat, die auch grosse Datenmengen möglichst schnell sortieren können. In diesem Kapitel werden wir uns mit solchen Verfahren (Sortieralgorithmen) beschäftigen und deren Laufzeit optimieren.

Definition 1. Ein *Sortieralgorithmus* ist ein Verfahren, das eine Liste von Elementen aus einer Menge sortiert. Wichtig ist dabei, dass diese Menge eine Ordnung besitzt (z.B. eine Rangordnung (Zahlen), eine alphabetische Ordnung, ...)!

Die Eingabe eines Sortieralgorithmus ist also eine durchmischte Liste von Elementen, die eine Ordnung besitzen. Der Sortieralgorithmus berechnet diese Ordung und gibt darauf eine sortierte Liste der Elemente aus. Damit unsere Listen von Elementen nicht zu aufwändig werden, beschränken wir uns im Folgeneden auf Listen von natürlichen Zahlen. Alle anderen möglichen Listen, wie Spielkarten oder Adressen, können auf Listen von natürlichen Zahlen zurückgeführt werden. Hier ein Beispiel für die Ein- und Ausgabe eines Sortieralgorithmus :

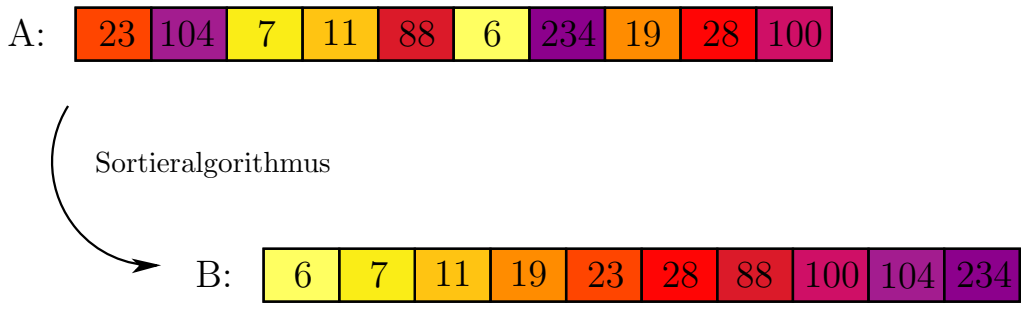


Abbildung 1: Beispiel für Input und Output eines Sortieralgorithmus

Aufgabe 4. *Gegeben sei folgende Liste von Elementen:*

$$a = [45 \mid 79 \mid 4 \mid 25 \mid 11 \mid 103 \mid 24 \mid 2 \mid 33 \mid 68].$$

(a) Wie sieht die sortierte Liste aus?

(b) Wie bist du vorgegangen um diese sortierte Liste zu bestimmen?

(c) Schreibe einen Algorithmus der das kleinste Element aus einer Liste $a = [a_1, a_2, \ldots, a_{10}]$ bestimmt.

Wenn du diese erste Aufgabe richtig gelöst hast, wird dir aufgefallen sein, dass es nicht möglich ist eine kleinste Zahl aus 10 beliebigen Zahlen zu ermittlen, ohne alle 10 Zahlen mindestens einmal durchzugehen. Ein schnellster Algorithmus geht genau einmal durch alle Zahlen: Man wählt zuerst die erste Zahl der Liste als Kandidaten für die kleinste Zahl. Dann geht man alle Elemente der Liste durch. Sobald man eine kleinere Zahl findet wird diese zur neuen kleinsten Zahl. Bis man am Schluss der 10 Zahlen wirklich die kleinste aller 10 Zahlen hat.

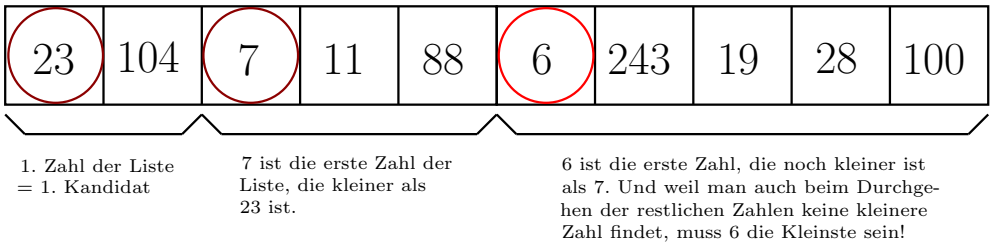


Abbildung 2: Liste von Zahlen mit 3 Kandidaten für eine kleinste Zahl

Aufgabe 5.
$a_1 = [15 \mid 8 \mid 10 \mid 11 \mid 6 \mid 20 \mid 3 \mid 21 \mid 17 \mid 1]$
$a_2 = [10 \mid 9 \mid 8 \mid 7 \mid 4 \mid 5 \mid 6 \mid 3 \mid 1 \mid 2]$
$a_3 = [5 \mid 6 \mid 7 \mid 8 \mid 1 \mid 2 \mid 3 \mid 4]$

(a) Wie viele Zahlen sind während dem Ablauf des oben beschriebenen Algorithmus Kandidaten für kleinste Zahlen?

(b) Wie viele solche Kandidaten, kann eine Liste von n Zahlen maximal haben? Und wie viele müssen es mindestens sein? (notiere zu beiden Fällen je ein Beispiel)

(c) Notiere ein Beispiel einer Liste, die genau zwei Kandidaten für kleinste Zahlen hat und bei der die 0 an der 5. Stelle steht.

Aufgabe 6. *Die Zahlen* $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ *werden in zufällige Reihenfolge gebracht. Jede Reihenfolge ist gleich wahrscheinlich.*

(a) Wie viele mögliche Reihenfolgen gibt es?

(b) Wie viele Reihenfolgen gibt es bei denen eine 0 *an der ersten Stelle steht?*

(c) Mit welcher Wahrscheinlichkeit braucht der oben erwähnte Algorithmus nur genau einen Kandidaten für eine kleinste Zahl?

(d) Mit welcher Wahrscheinlichkeit braucht er 10 *Kandidaten?*

Mit diesem Algorithmus für das Finden der kleinsten Zahl einer Liste, den wir jetzt in den letzten drei Aufgaben etwas unter die Lupe nahmen, können wir nun einen ersten einfachen Sortieralgorithmus schreiben: Wir suchen zuerst die kleinste Zahl der unsortierten Eingabeliste a und setzen sie auf die erste Stelle der Ausgabeliste b. Dann suchen wir die zweitekleinste in a und setzten sie auf die zweite Stelle in b; dann die drittkleinste ... und so weiter. Damit wir immer nach der kleinsten Zahl der ganzen Liste suchen können, setzen wir alle kleinsten Zahlen, die wir schon in b eingefügt haben, in a auf ∞. Setzen wir nämlich die kleinste Zahl auf ∞ und suchen darauf nochmals mit dem bekannten Algorithmus nach der kleinsten Zahl, so werden wir die ursprünglich zweitkleinste Zahl finden. Unser erster Sortieralgorithms sieht also wie folgt aus:

Algorithm 1: Erster Sortieralgorithmus

Input: Liste $a = [a_1, \ldots, a_n]$ mit n unterschiedlichen natürlichen Zahlen

Output: dazugehörige sortierte Liste $b = [b_1, \ldots b_n]$

$b := [0, \ldots, 0]$;
for ($i := 1$ *to* n) **do**
    Suche kleinste Zahl a_{kZ} aus a;
    $b_i := a_{kZ}$;
    $a_{kZ} := \infty$;
end

Aufgabe 7. *Gehe diesen ersten Sortieralgorithmus für das Beispiel* $a = [12 \mid 6 \mid 8 \mid 3 \mid 15 \mid 11 \mid 9]$ *durch und schreibe für jeden Durchlauf der For-Schleife* $(i = 1, 2, 3, \ldots, n)$ *a und b auf.*

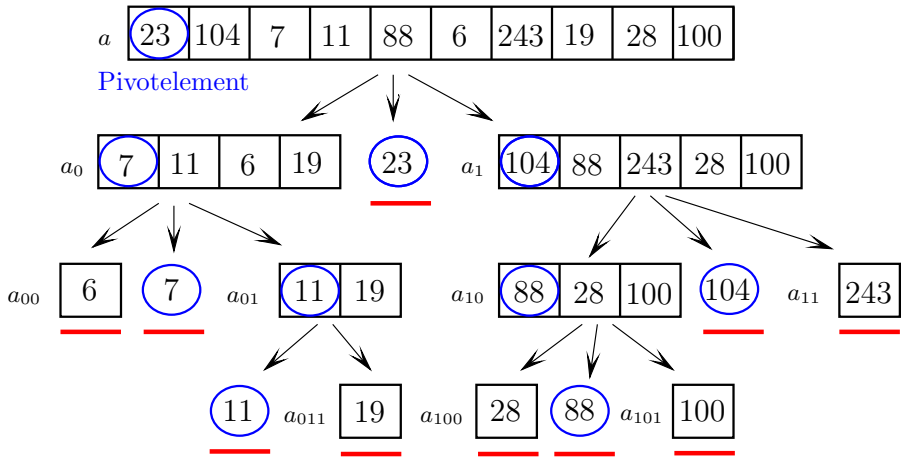
Aufgabe 8. *Wie viele Verlgeiche macht dieser Sortieralgorithmus um die ganze Liste a zu sortieren?*

3.2 Deterministischer Quicksort Algorithmus

In diesem Abschnitt werden wir einen schnelleren Sortieralgorithms kennen lernen: Den Quicksortalgorithmus. Dieser neue Sortieralgorithmus wird im Durchschnitt nur $n\log_2(n)$ Vergleiche brauchen statt n^2 (erster Sortieralgorithmus aus dem letzten Abschnitt). Die folgenede Tabelle veranschaulicht die Verbesserung der Laufzeit (Anzahl Vergleiche):

Anzahl Elemente	Erster Sortieralg.	Quicksortalg.	Verbesserung
n=10	100	33	3 mal schneller
n=100	10'000	664	15 mal schneller
n=1'000	1'000'000	9'966	100 mal schneller
n=10'000	100'000'000	132'877	753 mal schneller

Die Idee des Algorithmus ist folgende: Er teilt das Problem in immer mehr Teilprobleme, welche er dann, weil sie dadurch kleiner werden immer schneller lösen kann. Erhält der Quicksort-Algorithmus eine unsortierte Liste a von natürlichen Zahlen, so wählt er das erste Element der Liste und vergleicht es mit allen anderen Elementen. Alle Elemente, die kleiner sind als dieses erste Element, werden in eine Liste a_0 gespeichert und alle Elemente, die grösser sind als das erste Element werden in eine Liste a_1 gespeichert. Das erste Element der Liste, das man mit allen anderen vergleicht, nennt man *Pivotelement*. Nach diesem ersten Durchgang durch die Liste ist klar, dass zuerst alle Elemente von a_0 kommen, dann das Pivotelement und zum Schluss alle Elemente von a_1. Nun muss man aber noch die Reihenfolge der Elemente in a_0 und a_1 bestimmen. Das heisst man muss diese beiden Teillisten a_0 und a_1 sortieren. Dies geht nun aber um einiges schneller als die ganze Liste, denn die beiden Teillisten sind im Schnitt nur noch halb so lang wie die ursprüngliche ganze Liste. Um die beiden Teillisten zu sortieren, verwendet man einfach das selbe nochmals. Das heisst, man wird für die Liste a_0 das erste Element als neues Pivotelement wählen und es mit allen anderen Elementen von a_0 vergleichen und so wiederum zwei neue Listen a_{00} und a_{01} erhalten. Dasselbe macht man auch mit der anderen Teilliste a_1. So weiss man nach dem zweiten Durchlauf durch alle n Zahlen, dass zuerst jene Zahlen aus a_{00} kommen, dann das Pivotelement von a_0, dann alle Zahlen aus a_{01}, das Pivotelement von a, die Zahlen von a_{10}, das Pivotelement von a_1, und am Schluss die Zahlen aus a_{11}. Enthält eine dieser vier Teillisten $a_{00}, a_{01}, a_{10}, a_{11}$ noch mehr als ein Element, so teilt man diese weiter auf, bis alle Listen nur noch ein Element enthalten. Dieses rekursive Verfahren kann man mit Hilfe einer Baumdarstellung sehr schön veranschaulichen:



$\Rightarrow$ Output: $\{6, 7, 11, 19, 23, 28, 88, 100, 104, 243\}$

Abbildung 3: Baumdarstellung des Ablaufs von Quicksort

Das oben erklärte Verfahren des Quicksortalgorithmus kann man wie folgt kompakt zusammenfassen:

Algorithm 2: detQuicksort

Input: Liste $a = [a_1, \ldots, a_n]$ mit n versch. Zahlen aus $\mathbb{N}$

if $n = 1$ **then** (**return** a);
$pivot := a_1$;
$z_1 := 1$; $z_2 := 1$; (Zählvariablen)
for $(i := 2$ *to* $n)$ **do**
  if $a_i < pivot$ **then**
    $l_{z_1} := a_i$; $z_1 = z_1 + 1$;
  if $a_i > pivot$ **then**
    $r_{z_2} := a_i$; $z_2 = z_2 + 1$;
end
return *detQuicksort(l)*; **return** *pivot*; **return** *detQuicksort(r)*;

Dieser Algorithmus detQuicksort definiert immer das erste Element der Eingabe a als Pivotelement und spaltet danach a in l - *pivot* - r auf. In der Teilliste l speichert er alle Elemente von a die kleiner als das Pivotelement sind und in r alle jene die grösser sind. Die Teillisten l und r werden noch einmal in detQuicksort eingegeben und wieder als neue Liste a weiterbearbeitet, bis es nur noch Teillisten der Länge eins gibt.

Aufgabe 9.

(a) Gehe den Quicksortalgorithmus für

$$a_1 = [4 \mid 2 \mid 6 \mid 1 \mid 8 \mid 7 \mid 5 \mid 3]$$
$$a_2 = [1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8]$$

durch und skizziere die Baumdarstellung des Ablaufs.

(b) Welche Zahlen werden beim Durchlauf von a_1 mit der Zahl 4 verglichen? Und welche werden mit der Zahl 7 verglichen?

(c) Wie viele Vergleiche macht Quicksort bei diesen zwei Beispielen a_1 und a_2 insgesamt?

(d) Skizziere zusätzlich noch die Baumdarstellung des Aublaufs von Quicksort für $a_3 = [6 \mid 1 \mid 8 \mid 5 \mid 4 \mid 3 \mid 7 \mid 2]$. Warum werden bei a_3 die Zahlen 3 und 5 miteinander verglichen und bei a_1 nicht?

Gibt es gute und schlechte Listen für Quicksort?
Wie dir vielleicht bereits in Aufgabe 6 aufgefallen ist, brauchte der Quicksortalgrorithmus für das Sortieren von a_2 viel mehr Vergleiche als für das Sortieren von a_1. Ganz allgemein ist der Quicksortalgorithmus genau dann am schnellsten, wenn die beiden Teillisten, die er jeweils durch das Vergleichen mit dem Pivotelement erstellt, möglichst gleich gross sind. Ist hingegen die Liste aus n Zahlen bereits sortiert, so enthält die eine Liste des deterministischen Quicksortalgorithmus beim Aufspalten keine einzige Zahl und die andere dafür alle bis auf das Pivotelement. In diesem schlechten Fall macht der deterministische Quicksort, wie wir gleich nachrechnen werden, viel mehr Vergleiche! Das heisst, der Algortihmus läuft im schlechten Fall viel langsamer. Untersuchen wir diese beiden Fälle etwas genauer:
In einem guten Fall für Quicksort werden die Längen der Teillisten von Ebene zu Ebene immer mindestens halbiert (zur Veranschaulichung siehe Abbildung 4). Das heisst, in der ersten Ebene hat eine Teilliste maximal $\frac{n}{2}$ Elemente, in der zweiten Ebene noch maximal $\frac{n}{4}$ Elemente und in der k.-ten Ebene maximal $\frac{n}{2^k}$ Elemente. Wie viele Ebenen braucht in diesem Fall Quicksort, bis er nur noch Teillisten der Länge eins hat? Um das zu berechnen setzen wir die maximale Anzahl Elemente in der k.-ten Ebene gleich 1 und lösen diese Gleichung nach k auf:

$$\frac{n}{2^k} = 1 \Leftrightarrow n = 2^k \Leftrightarrow k = \log_2(n)$$

Der deterministische Quicksortalgorithmus braucht also im guten Fall nie mehr als $\log_2(n)$ Ebenen! Und da der Algorithmus pro Ebene nie mehr als n Vergleiche macht, kann man die Laufzeit des deterministischen Quicksorts im guten Fall auf $n\log_2(n)$ Vergleiche nach oben abschätzen.

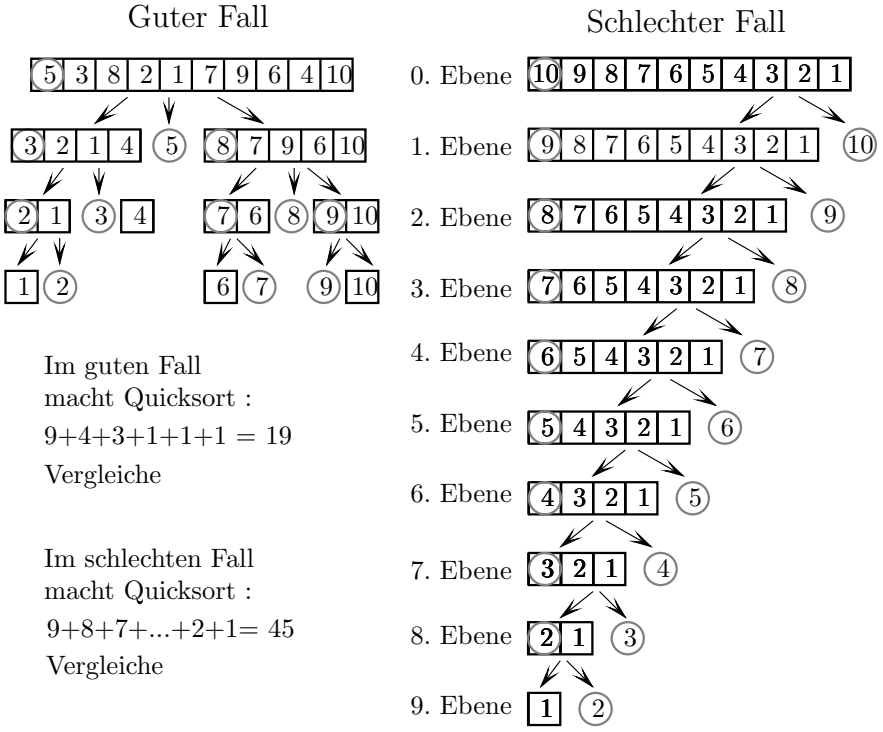


Abbildung 4: Beispiel einer guten und einer schlechten Liste für Quicksort

Im schlechtesten Fall von Quicksort ist die Eingabeliste eine bereits sortierte Liste. Die Teillisten werden von Ebene zu Ebene immer nur um ein Element kleiner. Der Algorithmus spaltet jeweils das Pivotelement ab und lässt sonst alle restlichen Elemente zusammen. So braucht Quicksort n Ebenen (inkl. der 0.-ten Ebene). In der 0-ten Ebene macht er $n-1$ Vergleiche, in der ersten Ebene macht er $n-2$ Vergleiche, ... und so weiter bis er dann am Schluss in der $n-1$-ten Ebene noch einen Verleich macht. Insgesamt macht Quicksort also:

$$1+2+3+\ldots+(n-1)=\sum_{k=1}^{n-1}k=\frac{n(n+1)}{2}$$

Vergleiche. Diese Laufzeit (Anzahl Vergleiche) ist bedeutend schlechter als jene des guten Falls. Für grosse Listen ist die Laufzeit vergleichbar schlecht

zur Laufzeit des einfacheren ersten Sortieralgorithmus aus dem letzten Abschnitt, denn für grosse n ist $\frac{n(n-1)}{2} \approx n^2$.
Im Durchschnitt braucht der deterministische Quicksortalgorithmus aber immer eine Anzahl Vergleiche, die sich für grosse Listen wie $n\log_2(n)$ verhält. Man sagt auch der deterministische Quicksortalgorithmus hat eine durchschnittliche Laufzeit von $O(n\log_2(n))$ (d.h. maximal von der Ordnung $n\log_2(n)$). Dazu aber mehr im nächsten Abschnitt.

3.3 Randomisierter Quicksort Algorithmus

Wir haben uns im letzten Abschnitt mit dem schlechten Fall des deterministischen Quicksort auseinander gesetzt. Wenn man diesem deterministischen Quicksortalgorithmus eine sortierte Liste als Eingabe gibt, so läuft er nicht schneller als der einfache erste Sortieralgorithmus! Das heisst, jemand der sich als Ziel eine Sabotage des deterministischen Quicksortalgorithmus vornimmt, weiss genau, dass man dazu nur eine sortierte Liste eingeben muss. Dies wollen wir nun verbessern! Ein Algorithmus, der bei der Eingabe des gewünschten Resultats am längsten rechnet, kann nicht unser Endresultat sein. Die Idee zur Verbesserung des Algorithmus ist, wie beim Codeknacker, das Einbauen des Zufalls! Anstatt, dass wir immer das erste Element der Liste als Pivotelement wählen, wählen wir nun neu das Pivotelement zufällig. Diese Abänderung des Algorithmus kostet uns nicht viel! Wir müssen nur eine Zeile änderen:

Algorithm 3: randomQuicksort

Input: Liste $a = [a_1, \ldots, a_n]$ mit n versch. Zahlen aus $\mathbb{N}$

if $n = 1$ **then** (**return** a);
$r :=$ zufällige Zahl aus $\{1, \ldots, n\}$;
$pivot := a_r$;
$z_1 := 1$; $z_2 := 1$; (Zählvariablen)
for ($i := 2$ *to* n) **do**
    if $a_i < pivot$ **then**
        $l_{z_1} := a_i$; $z_1 = z_1 + 1$;
    if $a_i > pivot$ **then**
        $r_{z_2} := a_i$; $z_2 = z_2 + 1$;
end
return *detQuicksort(l)*; **return** *pivot*; **return** *detQuicksort(r)*;

Werden die Pivotelemente zufällig gewählt, so hängt die Anzahl Vergleiche für das Soriteren nicht mehr von der Eingabeliste ab! Eine sortierte Liste

kann in diesem Fall genau gleich schnell sortiert werden, wie eine beliebige andere Liste. Ein Gegner des Algorithmus hat also keinen Vorteil mehr! Egal in welcher Reihenfolge er die Elemente in die Liste einträgt, der Zufall bestimmt die Dauer der Laufzeit. Es kann passieren, dass der randomisierte Algorithmus ganz ungünstige Wahlen des Pivotelements trifft, so dass er zuerst die grösste Zahl als Pivot wählt, dann die zweitgrösste, ... und so weiter. Diesen Fall wirst du dir in der nächsten Aufgabe noch etwas genauer anschauen. Hauptziel dieses Kapitels wird die Berechnung der zu erwartenden Laufzeit des randomisierten Quicksortsalgorithmus sein! Wie viele Vergleiche wird dieser Algorithmus im Durchschnitt gebrauchen? Davor nun aber noch ein paar allgemeine Aufgaben zum randomisierten Quicksort, die du dir als „Aufwärmung“ in dieses Thema anschauen solltest:

Aufgabe 10. *Die Eingabe des rand. Quicksort sei eine unsortierte Liste mit den Zahlen* $\{1, 2, \ldots, 10\}$. *Wie gross ist die Wahrscheinlichkeit, dass der schlechteste Fall eintritt? Das heisst mit welcher Wahrscheinlichkeit wählt der Algorithmus die Pivotelemente in sortierer Reihenfolge (also entweder* $1 - 2 - 3 - \ldots - 10$ *oder* $10 - 9 - 8 - \ldots 1$*)?*

Aufgabe 11. *Der rand. Quicksortalgorithmus erhält als Eingabeliste:* $[7 \mid 4 \mid 1 \mid 2 \mid 3 \mid 8 \mid 9 \mid 5 \mid 6 \mid 10]$. *Wie gross ist die Wahrscheinlichkeit für folgenden in der Baumdarstellung gezeichneten Sortierablauf?*

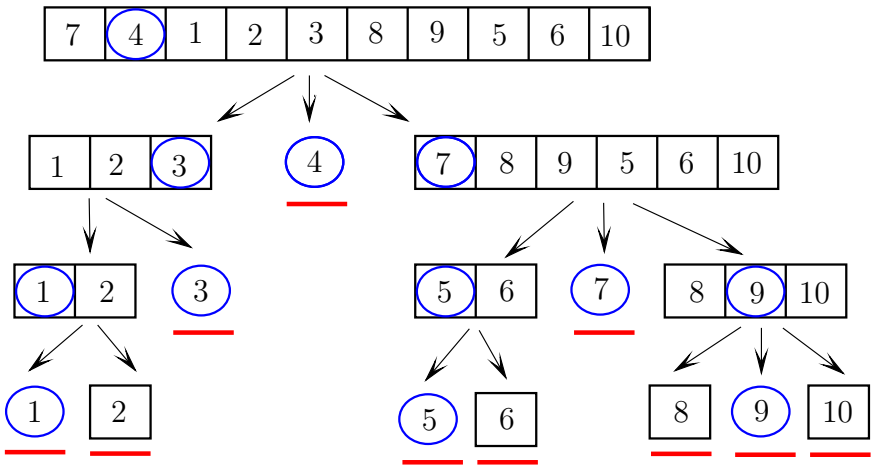


Abbildung 5: Baumdarstellung des Ablaufs

Tipp: Mit welcher Wahrscheinlichkeit wird die Zahl 4 *am Anfang aus der* 10*-elementigen Liste zum Pivot gewählt? Und mit welcher Wahrscheinlichkeit wird die Zahl* 3 *in der kleineren* 3*-elementigen Teilliste zum Pivot gewählt?*

Berechne also zuerst alle Wahrscheinlichkeiten dieser Pivotwahlen für die jeweiligen Teillisten. Was muss man mit diesen einzelnen Wahrscheinlichkeiten tun, um die gesamte gesuchte Wahrscheinlichkeit zu erhalten?

Aufgabe 12. *Die Eingabeliste sei wieder dieselbe, wie in Aufgabe 8. Finde einen Ablauf des randomisierten Quicksort, der mit möglichst hoher Wahrscheinlichkeit auftritt und zeichne den Ablauf in der Baumdarstellung.*

Laufzeitanalyse des randomisierten Quicksorts
Wir wollen nun zeigen, dass die durchschnittliche Laufzeit des randomisierten Quicksort von der Grössenordnung $O(n\log_2(n))$ ist (so wie der gute Fall des deterministischen Quicksorts). Dazu definieren wir die Zufallsvariable X, die für die Anzahl Vergleiche beim Ablauf des Randomisierten Quicksorts steht. Und das Ziel wird nun also die Berechung von $\mathrm{E}[X]$, die erwartete Anzahl Vergleiche, sein.

Die Hauptidee für das Ganze wird folgendes Teilproblem sein: Mit welcher Wahrscheinlichkeit werden zwei bestimmte Zahlen der Eingabeliste im Verlaufe der Berechung des Algorithmus miteinander verglichen?
Bearbeite folgende Aufgabe als ersten Schritt zur Lösung dieses Problems:

Aufgabe 13.
Gegeben sei die unsortierte Liste $[7 \mid 2 \mid 5 \mid 4 \mid 3 \mid 1 \mid 6]$. *Der randomisierte Quicksort wählt folgende zufällige Pivotelemente in der Reihenfolge:* $4, 2, 7, 5$. *Zeichne die Baumdarstellung des Ablaufs und fülle die weissen Felder der Tabelle aus: Gehe dazu alle Zahlenpaare* (i, j) *mit* $i \neq j$ *durch und überlege dir*

X_{ij}	1	2	3	4	5	6	7
1							
2							
3							
4							
5							
6							
7							

X_{ij}	1	2	3	4	5	6	7
1		X_{12}	X_{13}	X_{14}	X_{15}	X_{16}	X_{17}
2			X_{23}	X_{24}	X_{25}	X_{26}	X_{27}
3				X_{34}	X_{35}	X_{36}	X_{37}
4					X_{45}	X_{46}	X_{47}
5						X_{56}	X_{57}
6							X_{67}
7							

Abbildung 6: Welche Zahlenpaare werden miteinander verglichen?

ob diese beiden Zahlen i und j jemals beim Sortieren miteinander verglichen wurden. Wenn das der Fall ist schreibe in der Tabelle an der entsprechenden Stelle (X_{ij}) eine 1*. Werden die beiden Zahlen nie miteinander verglichen, dann schreibe eine* 0*.*

Allgemein gilt also in dieser Schreibweise aus Aufgabe 10 für i, j mit $i < j$:

$$X_{ij} = \begin{cases} 1 & \text{, die beiden Zahlen } i \text{ und } j \text{ werden miteinander verglichen} \\ 0 & \text{, sonst} \end{cases}$$

Bemerkung 1. Zählt man alle Einträge X_{ij} der Tabelle zusammen, so erhält man gerade die gesamte Anzahl Vergleiche, die der rand. Quicksort für das Sortieren der Liste benötigt.

Wann werden zwei Zahlen miteinander verglichen? Betrachten wir nochmals das Beispiel der letzten Aufgabe. Die Eingabeliste war [7 | 2 | 5 | 4 | 3 | 1 | 6] und das erste Pivotelement war die Zahl 4. Die Zahl 4 wurde somit mit allen anderen Zahlen verglichen. Die Eingabeliste wird dabei in zwei Teillisten geteilt: [2 | 3 | 1] und [7 | 5 | 6]. Die beiden Pivotelemente 2 und 7 der ersten Ebene werden aber nur noch mit jenen Elementen verglichen, die sich zu diesem Zeitpunkt noch zusammen in der selben Teilliste befinden. Zwei Zahlen, die einmal durch ein Pivotelement getrennt werden, wie hier zum Beispiel die 6 und 3, können später nie mehr miteinander verglichen werden. Etwas kürzer formuliert: Zwei Zahlen i und j werden genau dann miteinander verglichen, wenn eine Zahl davon zum Pivotelement gewählt wird und sich die andere zu diesem Zeitpunkt noch in der gleichen Teilliste befindet.
Wie lange befinden sich zwei Zahlen in der gleichen Teilliste? Betrachte dazu in den folgenden beiden Beispielen die Zahlen 2 und 4:

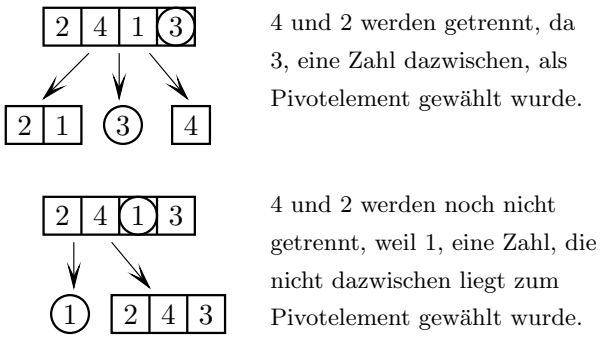


Abbildung 7: Wann bleiben 2 und 4 in der selben Teilliste?

Zwei Zahlen i und j $(i < j)$ bleiben also immer so lange in der selben Teilliste, bis eine Zahl aus $\{i, \ldots, j\}$ zum Pivotelemt gewählt wird.

Aufgabe 14. *Folgende unsortierte Liste wird dem rand. Quicksort als Eingabe gegeben:* [2 | 3 | 7 | 9 | 1 | 5 | 1 | 4 | 6 | 8].

(a) Bei welchen Wahlen des ersten Pivotelementes ist bereits klar, dass die Zahlen 2 *und* 7 *nie miteinander verglichen werden?*

(b) Wie gross ist somit die Wahrscheinlichkeit, dass nach der zufälligen Wahl des ersten Pivotelementes bereits klar ist, dass 2 *und* 7 *nie miteinander verglichen werden?*

Aufgabe 15. *Betrachte immer noch die selbe unsortierte Liste, wie in der letzten Aufgabe. Der randomisierte Quicksort wählt folgende Pivotelemente in der gegebenen Reihenfolge:*

(i) $8, 4, 7, 5, 2$

(ii) $9, 1, 7, 5, 3$

In welchen der oben beschriebenen Fälle werden die beiden Zahlen 2 *und* 7 *miteinander verglichen?*

Wenn wir also die Wahrscheinlichkeit berechnen wollen, dass zwei bestimmte Zahlen i und j $(i < j)$ aus einer unsortierten Liste mit den n natürlichen Zahlen $\{1, 2, \ldots, n\}$ miteinander verglichen werden, dann müssen wir die Wahrscheinlichkeit berechnen, dass i oder j zum Pivot gewählt wird, bevor eine Zahl k mit $i < k < j$ zum Pivot wird. Diese Wahrscheinlichkeit entspricht gerade einer bedingten Wahrscheinlichkeit. Nämlich der bedingten Wahrscheinlichkeit, dass die Zahl i oder j zum Pivot gewählt wird (=: Ereignis A), unter der Annahme, dass das Ereignis B: „keine Zahl k zwischen i und j wurde vorher zum Pivot gewählt“ bereits eingetroffen ist ($\Rightarrow$ Prob$[A|B]$). Alle Pivotwahlen ausserhalb des Zahlenbereichns $\{i, \ldots, j\}$ sind von den beiden Ereignissen A und B unabhängig! Wird eine Zahl kleiner i oder grösser j als Pivot gewählt, so gibt dies noch überhaupt keine Information darüber, ob später die Zahl i oder j vor den dazwischen liegenden Zahlen k zum Pivot gewählt wird. Es genügt somit nur den Bereich $\{i, \ldots, j\}$ zu betrachten. Wie gross ist die Wahrscheinlichkeit, dass man bei einer zufälligen gleichmässig verteilten Wahl aus $\{i, \ldots, j\}$ die Zahl i oder j wählt? Dies können wir mit der bekannten Formel: „Anzahl günstige Wahlen über Anzahl mögliche Wahlen“ einfach berechnen: Es gibt 2 günstige Wahlen, nämlich i und j und insgesamt $|\{i, \ldots, j\}| = j - i + 1$ mögliche Wahlen. Das heisst die Wahrscheinlichkeit, dass i oder j vor den dazwischenliegenden Zahlen zum Pivot gewählt wird ist $= \frac{2}{j-i+1}$.

Satz 1. Seien i und j zwei Zahlen ($1 \leq i < j \leq n$) aus einer unsortierten Liste mit den natürlichen Zahlen $\{1, 2 \ldots, n\}$ und sei X_{ij} das Ereignis, dass die beiden Zahlen i und j während dem Ablauf des randomisierten Quicksortalgorithmus miteinander verglichen werden, dann gilt:

$$\text{Prob}[X_{ij} = 1] = \frac{2}{j - i + 1}$$

Aufgabe 16. *Wie gross ist die Wahrscheinlichkeit, dass bei der Eingabeliste* [10 | 2 | 3 | 5 | 4 | 9 | 6 | 1 | 8 | 7 | 10] *der randomisierte Quicksort folgende Zahlenpaare miteinander vergleicht:*

(i) 4 *und* 9*?*

(ii) 6 *und* 7*?*

Aufgabe 17. *Sei nun die Eingabeliste* [98 | 35 | 3 | 21 | 77 | 76 | 2 | 12 | 44]*, also nicht mehr eine Liste mit Zahlen aus* $\{1, \ldots, n\}$*. Hast du eine Idee, wie man dann trotzdem noch die Wahrscheinlichkeit, dass 12 mit 77 verglichen wird, berechnen kann? Versuche diese Liste so abzuändern, dass du danach trotzdem noch mit Satz1 die gesuchte Wahrscheinlichkeit berechnen kannst.*

Kehren wir zum ursprünglichen Problem zurück:
Was ist die zu erwartende Anzahl Vergleiche, die der randomisierte Quicksort für das Sortieren einer Liste mit den n Zahlen aus $\{1, 2, \ldots, n\}$ durchführen muss? Was ist $\text{E}[X]$?

Wir benutzen nun die Idee aus der Bemerkung 1. Wir gehen alle Zahlenpaare (i, j) mit $i < j$ durch und zählen für jedes Paar bei dem i und j verglichen werden eins dazu. So erhalten wir, gerade die gesamte Anzahl Vergleiche. Etwas mathematischer formuliert heisst das, dass wir nun nicht mehr die Zufallsvariable X betrachten sondern alle Zufallsvariablen X_{ij} mit $1 \leq i < j \leq n$, die wir bereits wie folgt definiert haben:

$$X_{ij} = \begin{cases} 1 & \text{, die beiden Zahlen } i \text{ und } j \text{ werden miteinander verglichen} \\ 0 & \text{, sonst} \end{cases}$$

Wobei nach Bemerkung 1 also gilt: $\sum_{1 \leq i < j \leq n} X_{ij} = X$ (Das heisst: Zählt man alle Zufallsvariablen X_{ij} über alle möglichen i und j zusammen, so erhält man gerade die Zufallsvariable X)

Mit der Linearität des Erwartungswertes (d.h. mit der Formel: $\mathrm{E}[aX + bY] = a\mathrm{E}[X] + b\mathrm{E}[Y]$) gilt nun:

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}[\sum_{1\le i<j\le n} X_{ij}] && \{\text{nach Bemerkung 1}\}\\
&= \sum_{1\le i<j\le n} \mathrm{E}[X_{ij}] && \{\text{nach der Linearität des Erwartungswertes}\}\\
&= \sum_{1\le i<j\le n} (\mathrm{Prob}[X_{ij}=1]\cdot 1 + \mathrm{Prob}[X_{ij}=0]\cdot 0)\\
&\quad \{\text{Def. E}[X_{ij}] \text{ wobei } X_{ij} \text{ nur 1 oder 0 sein kann}\}\\
&= \sum_{1\le i<j\le n} \mathrm{Prob}[X_{ij}=1]\\
&= \sum_{1\le i<j\le n} \frac{2}{j-i+1} && \{\text{ nach Satz 1}\}
\end{aligned}$$

Das heisst der Erwartungswert von X entspricht der Summe aller Wahrscheinlichkeiten $\frac{2}{j-i+1}$ für i und j zwischen 1 und n mit $i < j$. Man geht also alle Zahlenpaare (i, j) der zu sortierenden Menge $\{1, 2, \ldots, n\}$ genau einmal durch, berechnet die Wahrscheinlichkeit, dass diese beiden Zahlen i und j miteinander verglichen werden ($\frac{2}{j-i+1}$) und addiert diese zusammen! Diese Summe wollen wir nun aber nicht exakt berechnen. Wir wollen sie abschätzen. Wir wollen eine obere Grenze für den Erwartungswert $\mathrm{E}[X]$ berechnen. Dann können wir damit festhalten, dass der randomisierte Quicksortalgorithmus nie mehr als so viele Vergleiche machen wird (Laufzeit $\le \ldots$).

Satz 2.

$$\sum_{1\le i<j\le n} \frac{2}{j-i+1} \le 2nH_n,$$

wobei H_n die n-te Harmonische Zahl bezeichnet (Summe aller reziproken natürlichen Zahlen bis $n = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$)

Beweis von Satz 2
Um diese Ungleichung zu zeigen, wollen wir zuerst einmal diese Summe umschreiben. Wir wollen nicht mehr über die verschiedenen Zahlenpaare (i, j) mit $1 \le i < j \le n$ summieren, sondern über deren Abstand $d := j - i$:

$$\sum_{1\le i<j\le n} \ldots \quad \Longrightarrow \quad \sum_{d=1}^{n-1} \ldots$$

Dazu müssen wir uns überlegen wie viele Paare welchen Abstand d haben:

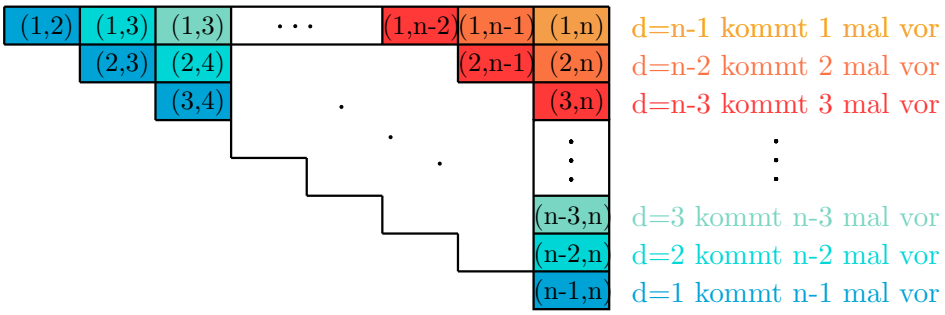


Abbildung 8: Umschreiben der Summe

Aus dieser Abbildung können wir erkennen, dass es genau ein Paar gibt, das den grossen Abstand $n-1$ hat. Paare mit Abstand $n-2$ gibt es aber bereits 2 und für den Abstand $n-3$ gibt es 3 Paare. Umso kleiner man den Abstand macht, desto mehr solche Zahlenpaare (i, j) kann man finden. Betrachtet man das Ganze noch etwas genauer so kann man feststellen, dass die Anzahl Zahlenpaare zum Abstand d gerade $n-d$ ist!
Wenn wir die Summe $\sum_{1\leq i<j\leq n} \frac{2}{j-i+1}$ wie oben erwähnt umschreiben wollen, dann müssen wir also alle i und j geschickt durch d ersetzten, so dass wir am Schluss immer noch dasselbe berechnen! Die Zahl $j-i$, die bei der Summe im Nenner steht, entspricht aber gerade unserem Abstand d vom Zahlenpaar (i, j). Das heisst wir können dies durch d ersetzten:

$$\frac{2}{j-i+1} \quad \Longrightarrow \quad \frac{2}{d+1}$$

Dann sollten wir noch folgendes beachten: In der alten Summe haben wir über jedes Paar (i, j) einzeln summiert $(1 \leq i < j \leq n)$. Wir hatten somit $n \cdot (n-1)$ Summanden. In der neuen Summe haben wir jedoch nur noch $n-1$ Summenden ($d = 1$, $d = 2$, …, $d = n-1$). Wir fassen nämlich alle Paare mit demselbem Abstand zusammen! Deshalb müssen wir die Zahl $\frac{2}{d+1}$ in der Summe noch mit der Anzahl solcher Paare multiplizieren. Die Anzahl solcher Paare haben wir uns bereits überlegt: Zum Abstand d gibt es gerade $n-d$ Paare (i, j) mit diesem Abstand. Die vollständige Umschreibung der Summe lautet deshalb:

$$\sum_{1\leq i<j\leq n} \frac{2}{j-i+1} \quad \Longrightarrow \quad \sum_{d=1}^{n-1} \frac{2}{d+1} \cdot (n-d)$$

Somit gilt:

$$\begin{aligned}
\sum_{1\le i<j\le n} \frac{2}{j-i+1} &= \sum_{d=1}^{n-1} \frac{2}{d+1}\cdot(n-d) \\
&\le \sum_{d=1}^{n-1} (\frac{2}{d+1})\cdot n \qquad \{(n-d)\le n\} \\
&= (\frac{2}{2}+\frac{2}{3}+\frac{2}{4}+\ldots+\frac{2}{n})\cdot n \\
&= 2n\cdot(\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\ldots+\frac{1}{n-2}+\frac{1}{n-1}+\frac{1}{n}) \\
&\le 2n\cdot(1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n-1}+\frac{1}{n}) \\
&= 2n\cdot H_n
\end{aligned}$$

□

Aus Satz 2 folgt nun, dass die zu erwartende Anzahl Vergleiche $\mathrm{E}[X] \le 2nH_n$ ist. Was sollen wir uns aber unter H_n vorstellen? Wie gross wird diese Harmonische Zahl H_n, wenn n grösser wird? Eine bekannte Abschätzung hilft uns dabei weiter. Sie besagt nämlich:

$$H_n = \sum_{k=1}^{n} \frac{1}{k} = \frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n} \le ln(n)+1.$$

Diese Abschätzung wollen wir hier nicht beweisen. Wir wollen sie in der folgenden Tabelle anhand gewisser Werte von n überprüfen (Die Werte sind jeweils auf zwei Stellen genau angegeben):

Grösse von n	H_n	$ln(n)+1$	$(ln(n)+1)-(H_n)$
n=1	1.00	1.00	0.00
n=2	1.50	1.69	0.19
n=5	2.28	2.60	0.33
n=10	2.93	3.30	0.37
n=50	4.50	4.91	0.41
n=100	5.19	5.61	0.42
n=1000	7.49	7.91	0.42

Die Werte der Harmonischen Zahl H_n bleiben immer kleiner als $ln(n)+1$. Gleichzeitig kann man aber auch beweisen, dass der Unterschied zwischen $ln(n)+1$ und H_n nie grösser wird als 0.58. Daraus folgt, dass die Harmonische

Zahl H_n wie $ln(n)$ wächst für grösser werdende n. Das heisst H_n ist von der Grössenordung $O(log(n))$. So ist also auch unsere zu erwartende Anzahl Vergleiche, die der randomisierte Quicksort macht, von der Grössenordnung $O(log(n))$. Und es folgt:

Satz 3. Der randomisierte Quicksortalgorithmus hat eine durchschnittliche Laufzeit von der Grössenordnung $O(n \log(n))$

Aufgabe 18.
Sei [3 | 5 | 7 | 2 | 1 | 6 | 4 | 8] *eine Eingabeliste für den rand. Quicksort. Und sei* Y *eine Zufallsvariable, die für die Anzahl Vergleiche des rand. Algorithmus von der Zahl* 4 *mit einer beliebigen anderen steht. Berechne E[Y].*
Tipp: Schreibe die Zufallsvariable Y *wieder mit Hilfe von Zufallsvariablen* X_{ij}.

4 Matrixmultiplikationstest

4.1 Einführung / Repetition Matrizen

Was ist eine Matrix? „The Matrix“ bezeichnet nicht nur einen bekannten Science-Fiction-Film, sondern auch ein wichtiges Grundkonzept der Mathematik! In der Mathematik ist eine Matrix ein Zahlenschema, das einer Tabelle sehr ähnlich kommt. Eine Matrix hat ebenfalls Zeilen und Spalten, die mit Zahlen gefüllt werden. Im Unterschied zur normalen Tabelle kann man aber Matrizen addieren oder miteinander multiplizieren, was man mit Tabellen so direkt nicht kann. Wie dies genau funktioniert möchten wir in diesem Unterabschnitt erarbeiten, bzw. repetieren.

Da man sich unter einer Matrix genau dann am besten etwas vorstellen kann, wenn man eine sieht, folgen hier ein paar Beispiele:

$$\mathbf{A} = \begin{pmatrix} 5 & 2 & 4 & 8 \\ 5 & 1 & 10 & 1 \\ 3 & 5 & 9 & 0 \end{pmatrix}, \qquad \mathbf{B} = \begin{pmatrix} 0 & 1 \\ 1 & 3 \end{pmatrix}, \qquad \mathbf{C} = \begin{pmatrix} 1 \\ 3 \\ 3 \end{pmatrix}$$

Wir werden Matrizen immer mit Grossbuchstaben bezeichnen. Die Matrix **A**, aus dem obigen Beispiel, hat drei Zeilen und vier Spalten. Man nennt eine solche Matrix deshalb auch 3×4-Matrix (gesprochen: „3 kreuz 4 Matrix“). Analog ist die Matrix **B** also eine 2×2-Matrix und die Matrix **C** eine 3×1-Matrix. Ganz allgemein wollen wir eine $m \times n$-Matrix, wie folgt definieren:

Definition 2. Eine $m \times n$-*Matrix* ist ein Zahlenschema mit m Zeilen und n Spalten:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

Die einzelnen Zahlen einer Matrix, die wir hier in der Definition mit den Variablen $a_{11}, a_{12}, \dots, a_{mn}$ bezeichnet haben, heissen auch *Einträge* der Matrix.

Aufgabe 19. *Schreibe zu jedem der drei angegebenen Matrix-Typen je ein konkretes Beispiel auf:*

(a) 5×2-*Matrix* *(b)* 3×3-*Matrix* *(c)* 1×4-*Matrix*

Aufgabe 20. *Du erhälst folgende Matrix:*

$$\mathbf{A} = \begin{pmatrix} 10 & 0 & 3 & 6 & 2 \\ 3 & 0 & 1 & 8 & 3 \\ -2 & 2 & 5 & 4 & 9 \\ 5 & 1 & 7 & 11 & 1 \end{pmatrix}$$

(a) Was für ein Typ Matrix ist das? (?×?-Matrix)

(b) Bestimme die Einträge: a_{11}, a_{24} und a_{35}.

(c) Was für Einträge stehen in der 3. Zeile und welche in der 2. Spalte?

Matrizen, die gleich viele Spalten wie Zeilen haben, nennt man auch *quadratische Matrizen* ($n \times n$-Matrizen). So war zum Beispiel die Matrix $\mathbf{B}$ ganz am Anfang dieses Abschnittes eine solche Matrix, nämlich eine quadratische Matrix der Grösse 2 (2×2-Matrix). Auch einen eigenen Namen erhalten Matrizen die nur eine Zeile, oder nur eine Spalte haben: Man nennt sie *Vektoren*. So war bei uns die Matrix $\mathbf{C}$ ein Vektor der Grösse 3 (einen Vektor mit 3 Einträgen).

Nun blieben wir bis jetzt auf einer sehr theoretischen und formalen Ebene von Matrizen. Können wir damit bereits einen Anschluss an unser tägliches Leben ziehen? Wie folgendes Beispiel zeigt, können wir Tabellen aus unserem Alltag in eine Matrix umschreiben:

Zeugnisnotentabelle der Klasse 4b

	Dt	Egl	Math	Phs	Bio	Gg
Martina	5	5.5	4.5	5.5	6	4.5
Julian	4.5	2.5	5.5	5	3.5	4.5
Ralph	3.5	5	4.5	5	4	4.5
Tanja	6	6	5	5.5	6	5
Wanda	4.5	4	5.5	3.5	6	3.5
Linda	4.5	5	4	3	4.5	5
Roger	5	5.5	6	6	5.5	6
Daniel	3	4	4.5	5	4.5	5
Kathrin	5.5	5	5	5	4	4.5
Lukas	5	4.5	3	4	5.5	5.5
Michael	4	3	3.5	4.5	4	5

$$\mathbf{A} = \begin{pmatrix} 5 & 5.5 & 4.5 & 5.5 & 6 & 4.5 \\ 4.5 & 2.5 & 5.5 & 5 & 3.5 & 4.5 \\ 3.5 & 5 & 4.5 & 5 & 4 & 4.5 \\ 6 & 6 & 5 & 5.5 & 6 & 5 \\ 4.5 & 4 & 5.5 & 3.5 & 6 & 3.5 \\ 4.5 & 5 & 4 & 3 & 4.5 & 5 \\ 5 & 5.5 & 6 & 6 & 5.5 & 6 \\ 3 & 4 & 4.5 & 5 & 4.5 & 5 \\ 5.5 & 5 & 5 & 5 & 4 & 4.5 \\ 5 & 4.5 & 3 & 4 & 5.5 & 5.5 \\ 4 & 3 & 3.5 & 4.5 & 4 & 5 \end{pmatrix}$$

Abbildung 9: Von der Tabelle zur Matrix

Dies alleine ist aber weiters nicht interessant! Es ist genau genommen nur eine leichte Umnotation. Um einen Vorteil der Matrizen gegenüber den normalen

Tabellen zu erlangen braucht es mehr! Wir wollen dazu die Addition und die Multiplikation von Matrizen etwas gnauer betrachten:

Addition von Matrizen
Damit man zwei Matrizen addieren kann müssen sie genau gleich gross sein! Ist das der Fall, so nimmt man die Einträge, die jeweils an der selben Stelle (selbe Zeile und selbe Spalte) stehen, und addiert diese zwei Zahlen. So ist die Summe zweier Matrizen wieder eine Matrix derselben Grösse.

Definition 3. Sind $\mathbf{A}$ und $\mathbf{B}$ zwei $m \times n$-Matrizen, so gilt:

$$\begin{aligned}\mathbf{A}+\mathbf{B} &= \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix} + \begin{pmatrix} b_{11} & \dots & b_{1n} \\ \vdots & & \vdots \\ b_{m1} & \dots & b_{mn} \end{pmatrix} \\ &= \begin{pmatrix} a_{11}+b_{11} & \dots & a_{1n}+b_{1n} \\ \vdots & & \vdots \\ a_{m1}+b_{m1} & \dots & a_{mn}+b_{mn} \end{pmatrix}\end{aligned}$$

Aufgabe 21. *An einem Sporttag werden fünf Gruppen A, B, C, D, E zu jeweils 4 Spielern gebildet. Es müssen drei Posten durchgeführt werden, in denen man jeweils bis zu 10 Punkte erreichen kann. Die untenstehenden Tabellen zeigen das Ergebnis der einzelnen Spieler. Schreibe die drei Tabellen in Matrizen* $\mathbf{P}$, $\mathbf{K}$ *und* $\mathbf{R}$ *um. Und berechne* $\mathbf{T} = \mathbf{P} + \mathbf{K} + \mathbf{R}$. *Was geben die Einträge der Matrix* $\mathbf{T}$ *an?*

Pfeil & Bogen schiessen

	A	B	C	D	E
Spieler 1	6	10	9	4	2
Spieler 2	8	5	10	7	8
Spieler 3	10	3	9	6	9
Spieler 4	7	5	4	7	6

Kickboardparcour

	A	B	C	D	E
Spieler 1	10	5	10	4	8
Spieler 2	10	7	8	9	6
Spieler 3	7	6	9	7	6
Spieler 4	6	8	5	7	9

Ringwerfen

	A	B	C	D	E
Spieler 1	7	5	7	4	6
Spieler 2	4	6	10	8	10
Spieler 3	9	3	9	7	9
Spieler 4	8	5	8	5	3

Abbildung 10: Punktetabellen Sporttag

Aufgabe 22. *Gegeben seien folgende Matrizen:*

$$\mathbf{A} = \begin{pmatrix} 2 & 4 \\ 3 & 1 \\ 5 & 2 \end{pmatrix}, \qquad \mathbf{B} = \begin{pmatrix} -1 & -4 \\ 0 & 3 \end{pmatrix}, \qquad \mathbf{C} = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 1 & 2 \end{pmatrix}$$

$$\mathbf{D} = \begin{pmatrix} -2 & 0 \\ 3 & 4 \\ 5 & 1 \end{pmatrix}, \qquad \mathbf{E} = \begin{pmatrix} 5 & 8 \\ 2 & 1 \end{pmatrix}$$

Berechne, falls möglich, die Summe:

(a) $\mathbf{A} + \mathbf{B}$ *(b)* $\mathbf{A} + \mathbf{D}$

(c) $\mathbf{B} + \mathbf{E}$ *(d)* $\mathbf{C} + \mathbf{E}$

Multiplikation von Matrizen
Matrizen können mit Zahlen multipliziert werden. In diesem Fall wird jeder Eintrag der Matrix mit dieser Zahl multipliziert. Das heisst das Produkt einer Zahl und einer Matrix ergibt wieder eine Matrix derselben Grösse.

Definition 4. Sei $\mathbf{A}$ eine $m \times n$-Matrix und c eine Zahl, so gilt:

$$c \cdot \mathbf{A} = c \cdot \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix} = \begin{pmatrix} ca_{11} & \dots & ca_{1n} \\ \vdots & & \vdots \\ ca_{m1} & \dots & ca_{mn} \end{pmatrix}$$

Aufgabe 23. *Gegegeben seien folgende Matrizen:*

$$\mathbf{A} = \begin{pmatrix} 3 & 2 \\ 5 & -1 \end{pmatrix}, \qquad \mathbf{B} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}$$

Berechne:

(a) $2\mathbf{B}$ *(b)* $2\mathbf{A} + 3\mathbf{B}$

(c) $2\mathbf{A} - \mathbf{B}$ *(*$= 2\mathbf{A} + (-1)\mathbf{B}$*)*

Etwas aufwändiger und anspruchsvoller wird nun aber die Multiplikation zweier Matrizen. Damit die Multiplikation zweier Matrizen möglich ist, muss die erste Matrix gerade so viele Spalten haben, wie die zweite Matrix Zeilen hat. Das heisst ist also $\mathbf{A}$ eine $m \times n$-Matrix und $\mathbf{B}$ eine $n \times l$-Matrix, so kann man das Produkt $\mathbf{A} \cdot \mathbf{B}$ berechnen. Das Produkt dieser beiden Matrizen ist dann eine $m \times l$-Matrix, nennen wir sie $\mathbf{C}$. Um den Eintrag c_{ij} dieser Produktematrix zu berechnen, braucht man die i-te Zeile von $\mathbf{A}$ und die j-te Spalte von $\mathbf{B}$. Man multipliziert den ersten Eintrag der i-ten Zeile von $\mathbf{A}$ mit dem ersten Eintrag der j-ten Spalte von $\mathbf{B}$, den zweiten Eintrag der i-ten Zeile von $\mathbf{A}$ mit dem zweiten Eintrag der j-ten Spalte von $\mathbf{B}$, ... u.s.w., bis man

das Produkt des n-ten Eintrag dieser Zeile von $\mathbf{A}$ und des n-ten Eintrags der Spalte von $\mathbf{B}$ auch noch berechnet hat. Dann zählt man diese n Produkte zusammen. Das ergibt uns dann den Eintrag c_{ij} der Produktematrix. Um alle Einträge c_{ij} der Produktematrix zu berechnen müssen wir also jede Spalte von $\mathbf{B}$ einmal mit jeder Zeile von $\mathbf{A}$ durchrechnen. Diesen Prozess des Berechnen eines einzelnen Eintrags c_{ij} der Produktematrix kann man wie folgt etwas veranschaulichen:

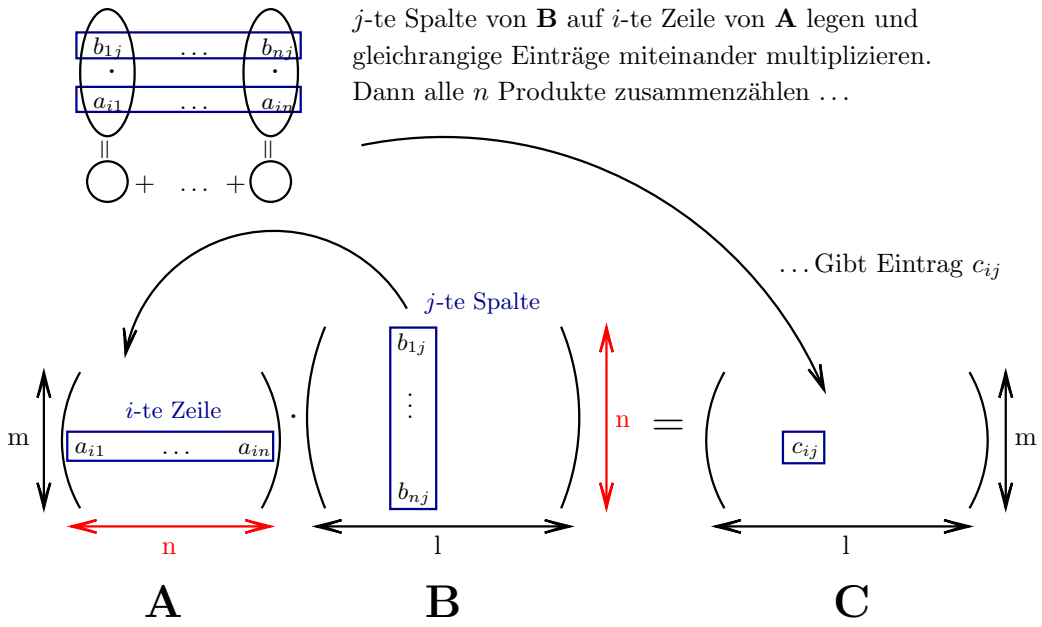


Abbildung 11: Multiplikation zweier Matrizen $\mathbf{A}$ und $\mathbf{B}$

Definition 5. Ist $\mathbf{A}$ eine $m \times n$-Matrix und $\mathbf{B}$ eine $n \times l$-Matrix, so kann man das Produkt $\mathbf{A} \cdot \mathbf{B}$ (=: $\mathbf{C}$) berechnen:

$$\mathbf{A} \cdot \mathbf{B} = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \dots & b_{1l} \\ \vdots & & \vdots \\ b_{n1} & \dots & b_{nl} \end{pmatrix} = \begin{pmatrix} c_{11} & \dots & c_{1l} \\ \vdots & & \vdots \\ c_{m1} & \dots & c_{ml} \end{pmatrix}$$

$$\text{Mit } c_{ij} = a_{i1}b_{1j} + \ldots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj} \text{ für } 1 \leq i \leq m \text{ und } 1 \leq j \leq l$$

Aufgabe 24. *Berechne:*

(a) $\begin{pmatrix} 2 & 3 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 4 & 0 \\ 1 & 1 \end{pmatrix}$

(b) $\begin{pmatrix} 1 & 1 & 2 & 3 \\ 0 & 0 & 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} 4 & 0 & 5 \\ 2 & 1 & -1 \\ 1 & 2 & -1 \\ 3 & 0 & 2 \end{pmatrix}$

Wir möchten nun noch etwas speziell auf Vektoren eingehen. Wie bereits erwähnt sind das Matrizen, die nur aus einer Spalte (*Spaltenvektoren*) oder nur aus einer Zeile (*Zeilenvektoren*) bestehen. Damit man die Vektoren klar von den restlichen Matrizen abgrenzen kann, werden wir sie ab nun mit Kleinbuchstaben bezeichnen:

$$u = \begin{pmatrix} u_1 & \dots & u_n \end{pmatrix} \text{ Zeilenvektor}, \qquad v = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} \text{ Spaltenvektor}$$

Multipliziert man eine Matrix mit einem Vektor oder umgekehrt, so ergibt das, wenn überhaupt möglich, wieder einen Vektor. Untersuche dies anhand der folgenden Aufgabe:

Aufgabe 25. *Gegeben seien eine* 3×3*-Matrix* $\mathbf{A}$*, ein Zeilenvektor* u *der Grösse* 3 *und ein Spaltenvektor* v *der selben Grösse:*

$$\mathbf{A} = \begin{pmatrix} 3 & 1 & 1 \\ 3 & 0 & 2 \\ 2 & 1 & 2 \end{pmatrix} \qquad u = \begin{pmatrix} 2 & 1 & -1 \end{pmatrix} \qquad v = \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix}$$

Berechne, falls möglich:

(a) $v \cdot \mathbf{A}$ *(b)* $\mathbf{A} \cdot v$ *(c)* $u \cdot \mathbf{A}$ *(d)* $\mathbf{A} \cdot u$

Aufgabe 26. *Sei* $\mathbf{T}$ *die Matrix aus der Sporttagaufgabe (Aufgabe 21). Berechne:*

$$\begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix} \cdot \mathbf{T}$$

Interpretiere die Einträge dieser Matrix!

Aufgabe 27. *Gegeben seien die Matrizen:*

$$\mathbf{A} = \begin{pmatrix} 2 & 1 \\ 1 & 3 \\ 1 & 0 \end{pmatrix}, \ \mathbf{B} = \begin{pmatrix} 4 & 2 & 1 \\ -1 & 0 & 0 \\ 1 & 1 & 2 \end{pmatrix}, \ \mathbf{C} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \ \mathbf{D} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{pmatrix}$$

Berechne:

(a) $\mathbf{B}(\mathbf{A} - \mathbf{D})$ *(b)* $(\mathbf{B} + \mathbf{C}) + \mathbf{BD}$ *(c)* $\mathbf{CB}$

Aufgabe 28. *Pascal, David, Alex und Philipp planen eine Party. Ihnen steht nur ein sehr knappes Budget für Snacks und Getränke zur Verfügung. Und alle haben unterschieliche Präferenzen von Lebensmitteln, die sie gerne einkaufen würden. Um eine Entscheidung zu treffen, vergleichen sie die gesamten Ausgaben der vier Listen bei den unterschiedlichen Einkaufsläden (siehe anschliessende Tabelle). Schreibe die Information der Tabelle in zwei Matrizen, so dass du diese mitainander multiplizieren kannst und die Produktematrix gerade die gewünschten Ausgabensummen enthält.*

	Von der Person gewünschte Anzahl Einheiten				Preis pro Einheit		
	Pascal	David	Alex	Philipp	Coop	Migros	Denner
Salzgebäck	5	2	4	3	2.10	2.30	2.00
Schokolade	0	6	3	0	2.20	1.30	1.20
Bisquitt	3	6	2	2	2.30	1.80	2.30
Brötli	4	2	4	6	2.30	2.00	2.20
Salat	2	0	2	6	2.50	2.20	2.40
Eis	2	5	0	2	3.50	3.80	4.20
Süssgetränke	4	6	4	2	2.50	2.20	1.80
Mineral	4	0	2	8	1.10	1.00	0.90

Abbildung 12: Preis- und Wunschvergleich für Partyeinkauf

4.2 Verifikation von Matrixmultiplikationen

Wie wir im letzten Abschnitt feststellen mussten ist die Multiplikation zweier Matrizen aufwändig. Man muss viele Teilprodukte der Matrizeneinträge berechnen und diese anschliessend noch aufsummieren. Wie können wir geschickt nachkontrollieren, ob unser Resultat stimmt? Auf diese Frage wollen wir in diesem Kapitel genauer eingehen. Damit das Ganze etwas anschaulicher wird, betrachten wir ein konkretes Problem:

Das Papeterie-Bestellungs-Problem: Ein Papeteriegeschäft muss zwei mal im Jahr eine Grossbestellung aufgeben. Dabei können je nach Trend der Nachfrage unterschiedliche Schwerpunkte bei der Artikelwahl gesetzt werden und allenfalls der Lieferant gewechselt werden. Die Aufgabe des Buchhalters ist es nun, verschiedene Bestellungsmöglichkeiten durchzurechnen und jeweils die totalen Kosten bei den unterschiedlichen Firmen zu bestimmen. Das macht er mit Hilfe einer Matrixmultiplikation:

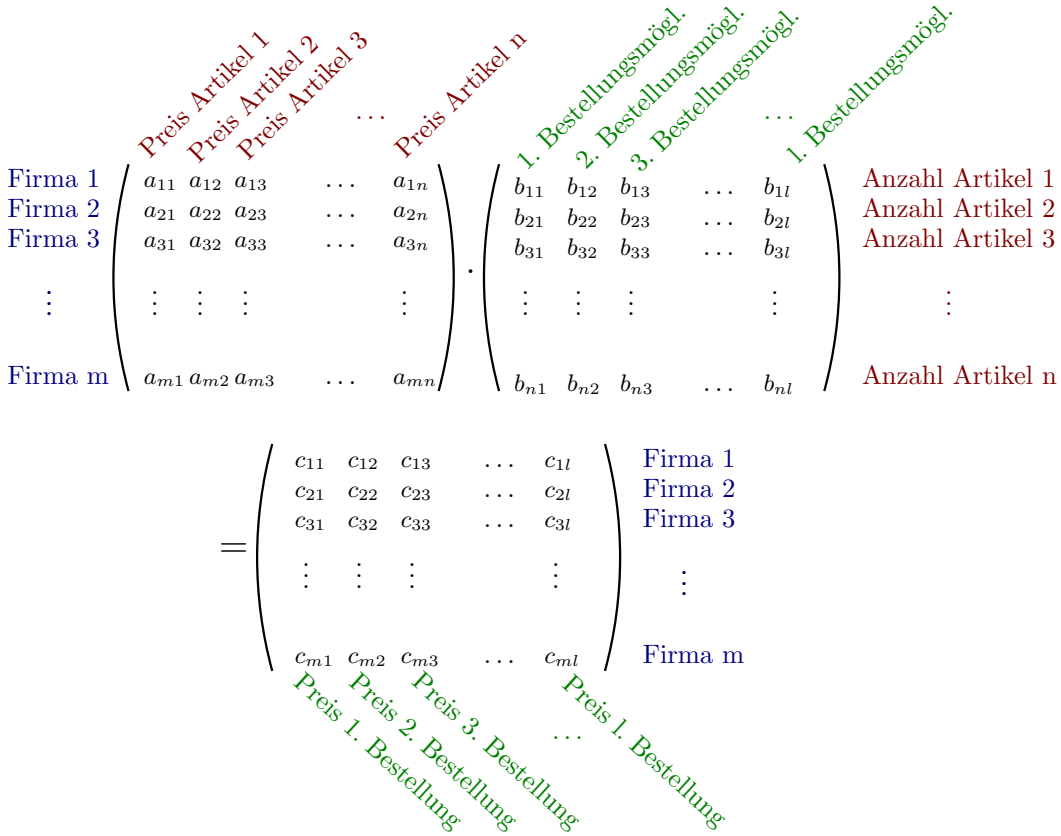


Abbildung 13: Papeterie-Bestellungs-Problem

Es gibt insgesamt n Artikel, von denen man in den l verschiedenen Bestellungsmöglichkeiten eine gewisse Anzahl auswählt und es hat m verschiedene Firmen, die alle diese Artikel liefern können. Dem Buchhalter sind alle Preise der Artikel von jeder Firma bekannt. Er schreibt die Preise der Firmen für die Artikel in die erste Matrix $\mathbf{A}$ *und die benötigte Anzahl der Artikel für die Bestellungsmöglichkeiten in eine zweite Matrix* $\mathbf{B}$*. Seine Aufgabe besteht somit nur noch aus der Multiplikation* $\mathbf{A} \cdot \mathbf{B}$*. Denn die berechnete Produktematrix, nennen wir sie* $\mathbf{C}$*, gibt dann genau die gewünschten Gesamtkosten der Bestungsmöglichkeiten bei den einzelnen Firmen an (siehe Abbildung 13).*
Der Chef der Papeterie ist darauf angewiesen, dass die berechneten Preise korrekt sind. Doch der Buchhalter ist nicht immer ganz unparteiisch und hatte auch schon seine Resultate zu Gunsten gewisser Frimen abgeändert. Wie kann der Chef das Resultat des Buchhalters überprüfen?

Damit der Chef sicher sein kann, dass alle Einträge der berechneten Matrix $\mathbf{C}$ stimmen, bleibt ihm auf dem deterministischen Weg nichts anderes übrig, als alle Einträge nachzurechnen. Das heisst er müsste die ganze Arbeit des Buchhalters noch einmal durchführen. Dies kann also nicht die gewünschte Lösung des Problems sein.

Können wir dies durch den Einbau von Zufall verbessern? Gibt es ein randomisiertes Verfahren für den Chef, bei dem er weniger rechnen muss? Überlegen wir uns dafür zuerst einmal wie viel der Buchhalter gerechnt hat:

Aufgabe 29. *Gegeben seien folgende Matrizenmultiplikationen:*

(a) $\begin{pmatrix} 2 & 1 \\ 4 & 1 \end{pmatrix} \cdot \begin{pmatrix} 3 & 1 \\ 1 & 1 \end{pmatrix}$

(b) $\begin{pmatrix} 2 & 1 & 1 \\ 2 & 1 & 3 \\ 1 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} 2 & 1 & 3 \\ 4 & 2 & 3 \\ 1 & 1 & 1 \end{pmatrix}$

Wie viele Multiplikationen müssen berechnet werden um diese Matrixprodukte zu bestimmen? (Die Produktematrix muss dazu nicht zwingend bestimmt werden.)

Damit für uns die Überlegungen mit dem Papeterieproblem nicht unnötig kompliziert werden, nehmen wir für die weiteren Überlegungnen an, dass die drei Variablen m,n und l (Anzahl Firmen, Artikel und Bestellungsmöglichkeiten) alle gleich gross sind ($= n$). Der Buchhalter musste also $n \cdot n = n^2$ Matrixeinträge für die Produktematrix $\mathbf{C}$ berechnen. Und da er für jeden Eintrag

der Produktematrix n Multiplikationen duchführen musste, hat er insgesamt $n \cdot n^2 = n^3$ Multiplikationen berechnet. Das heisst, bei dem vollständigen Überprüfen der Rechnung müsste auch der Chef n^3 Multiplikationsschritte durchführen.

Wir wollen nun ein randomisiertes Verfahren erarbeiten für das der Chef nur n^2 Multiplikationen berechnen muss, um zu entscheiden ob die Rechnung $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$ des Buchhalters korrekt ist. Dies ist ein enormer Zeitgewinn für die Nachkontrolle. Vor allem für grosse Matrizen:

Matrixgrösse $[n \times n]$-Matrix	Vollständiger Test [# Multiplikationen]	Rand. Verfahren [# Multiplikationen]	Rand. Verfahren ist ...
$n = 10$	1'000	100	10 mal schneller
$n = 50$	125'000	2'500	50 mal schneller
$n = 100$	1'000'000	10'000	100 mal schneller
$n = 500$	125'000'000	250'000	500 mal schneller

Abbildung 14: Geschwindigkeitsvorteil eines Rand. Testverfahrens

Das randomisierte Verfahren ist also immer n-mal schneller als das deterministische. Diesen Zeitgewinn müssen wir bezahlen. Das Verfahren kann uns nämlich mit einer gewissen Wahrscheinlichkeit ein falsches Resultat liefern. Das heisst, es kann passieren, dass uns der Algorithmus am Schluss angibt, dass $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$ sein soll, obwohl der Buchhalter in Wirklichkeit eine falsche Matrix $\mathbf{C}$ geliefert hat. Ein solcher Algorithmus läuft also wie ein Zufallsexperiment: Es gibt verschiedene vorgegebene Berechnungswege für welche sich der Algorithmus zufällig und gleichmässig verteilt entscheiden wird. Interessant ist dabei das Verhältnis zwischen den Anzahl möglichen Berechnungen, die zu einem korrekten Resultat führen und der totalen Anzahl möglichen Berechnungen. Dies gibt uns nämlich gerade die Wahrscheinlichkeit, dass ein solcher Algorithmus ein korrektes Resultat ausgibt.

Kann der Chef einen solchen Algorithmus überhaupt gebrauchen? Einen Algorithmus, der je nach dem den Buchhalter noch unterstützt und auch behauptet, das Resulatat sei korrekt, ohne, dass dies der Fall ist? Die Antwort daruaf ist: Ja! Es gibt nämlich einen relavanten Unterschied zwischen dem Algorithmus und dem Buchhalter: Lässt man den Buchhalter mehrmals seine Rechnung durchführen, so kann es gut sein, dass er jedes mal denselben Fehler macht, oder sogar willkürlich immer denselben Matrixeintrag abändert. Der Algorithmus hingegen läuft völlig zufällig und gleichmässig verteilt! Das

heisst, wenn man diesen mehrmals rechnen lässt, wird die Wahrscheinlichkeit grösser, dass er dabei einmal ein korrektes Resultat liefert. Untersuche dies in der folgenden Aufgabe selbst:

Aufgabe 30. *Ein randomisierter Algorithmus gibt mit der Wahrscheinlichkeit $\frac{1}{2}$ ein korrektes Resultat aus.*

(a) Man lässt den Algorithmus zwei mal laufen. Wie gross ist die Wahrscheinlichkeit, dass der Algorithmus zwei mal ein falsches Resulat liefert?

(b) Wie viel mal muss man den Algorithmus laufen lassen, damit er mit einer Wahrscheinlichkeit von 0.999 mindestens einmal ein korrektes Resultat bringt?

Diese Aufgabe zeigt uns also, dass sogar ein Algorithmus, bei dem die Hälfte der möglichen Berechnungen zu einem falschen Resultat führen, nach ein paar wenigen Wiederholungen mit hoher Wahrscheinlichkeit mindestens einmal ein korrektes Resultat liefert. Dies wird uns genügen, um mit einer gleich hohen Wahrscheinlichkeit die richtige Entscheidung ($\mathbf{A}\cdot\mathbf{B} = \mathbf{C}$ oder $\mathbf{A}\cdot\mathbf{B} \neq \mathbf{C}$) für den Papeteriechef zu fällen. Wie genau man diese Entscheidung trifft sehen wir später.

Entscheidend ist: Die Kosten für den Zeitgewinn (n mal schneller) durch den randomisierten Algorithmus werden also nicht so gravierend hoch sein! In der Praxis ist es oft wichtiger, schnell arbeiten zu können, als mit einer Wahrscheinlikeit von 0.001 einen Fehler zu machen.

Wir haben also gesehen, dass uns ein solches randomisiertes Verfahren weiterhelfen kann. Wie soll es aber funktionieren? Was ist die Idee dahinter? Man muss eine Äquivalenz überprüfen ($\mathbf{AB} \stackrel{?}{=} \mathbf{C}$), ohne jedes Detail (alle Matrixeinträge) beider Seiten genau zu berechnen und zu vergleichen. Die Methode, die dahinter steckt, nennt man auch Methode der Fingerabdrücke. So wie die Kriminalpolizei Fingerabdrücke verwendet um Täter voneinander zu unterscheiden, wollen wir Fingerabdrücke für unsere Matrizen ($\mathbf{AB}$ und $\mathbf{C}$) konstruieren, um zu entscheiden ob es sich um die selbe Matrix handelt oder nicht. Wir werden unseren Fingerabdruck mit Hilfe des Zufalls konstruieren. Der Fingerabruck unserer Matrix soll drei Punkte erfüllen:

(1.) Es soll möglich sein den Fingerabdruck einfach und schnell zu berechnen.

(2.) Der Fingerabdruck selbst soll genügend komprimiert sein, so dass man zwei Fingerabdrücke schnell miteinander vergleichen kann.

(3.) Der Fingerabdruck muss trotzdem möglichst viel relevante Informationen über die Matrizen enthalten, so dass das Ergebnis des Vergleichs von Fingerabdrücken mit genügend hoher Wahrscheinlichkeit korrekt ist.

In den folgenden beiden Abschnitten werden wir auf zwei Möglichkeiten eines solchen Fingerabrucks eingehen.

Stichproben-Fingerabdruck
Diese erste Methode eines zufälligen Fingerabdruckes ist recht intuitiv. Was macht man, wenn man eine Liste von Rechnungen bekommt und möglichst schnell entscheiden muss, ob diese Rechnungen stimmen, ohne, dass man die Zeit hat alle nachzurechnen? Man wird sich zufällig ein paar Rechnungen herauspicken und diese kontrollieren. Wenn man dabei bereits einen Fehler findet, kann man schon auf sicher antworten, dass nicht alles korrekt ist. Wenn man hingegen keinen Fehler gefunden hat, kann man nur mit einer gewissen Erfolgswahrscheinlichkeit darauf tippen, dass alle Rechnungen korrekt sind. Überlege dir das selbst in der nächsten Aufgabe:

Aufgabe 31. *Du erhälst 10 Rechnungen und musst entscheiden, ob alle korrekt sind. Du gehst wie oben beschrieben vor und wählst zufällig vier Rechnungen aus und rechnest sie nach. Findest du dabei einen Fehler so antwortest du „nicht alle Rechnungen sind korrekt“ und, wenn du keinen Fehler findest „alle Rechnungen sind korrekt“.*

(a) Angenommen vier der Rechungen sind falsch. Wie gross ist die Wahrscheinlichkeit, dass du keine falsche Rechnung findest und „alle Rechnungen sind korrekt“ antwortest?

(b) Wenn du genau so wie oben beschrieben vorgehst und dich selbst nicht verrechnest: Wie gross ist dann die Wahrscheinlichkeit, dass du „nicht alle Rechnungen sind korrekt“ antwortest, obwohl in Wirklichkeit doch alle Rechnungen stimmen?

Aus dieser Idee der zufälligen Überprüfung einer Stichprobe ergibt sich folgender Algorithmus für den Papeteriechef: Man wählt zufällig s Einträge von der Matrix $\mathbf{C}$ aus und überprüft diese. Gilt für einen Eintrag c_{ij} der Matrix $\mathbf{C}$: $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$ (siehe Definition 5), so ist dieser Eintrag korrekt. Der Algorithmus rechnet weiter. Findet er einen Eintrag c_{ij} für den diese Gleichheit nicht gilt, so bricht er ab und gibt $\mathbf{AB} \neq \mathbf{C}$ aus. Wenn er hingegen

in keinem der s ausgewählten Einträgen von $\mathbf{C}$ einen Fehler entdeckt, so antwortet er $\mathbf{AB} = \mathbf{C}$. Der Fingerabdruck ist in diesem Fall eine zufällige Stichprobe von s Einträgen der Matrix $\mathbf{C}$.

Algorithm 4: Stichproben-Test-Algorithmus

Input: drei $n \times n$-Matrizen $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$
Output: „$\mathbf{AB} = \mathbf{C}$“ oder „$\mathbf{AB} \neq \mathbf{C}$“

$s :=$ Anzahl Matrixeinträge, die man zufällig überprüfen will;
$A :=$ Menge von s zufällig ausgewählten Matrixeinträgen von $\mathbf{C}$;
for ($m := 1$ *to* s) **do**
    Sei der m-te Eintrag von $A = c_{ij}$ in $\mathbf{C}$: ;
    if ($\sum_{k=1}^{n} a_{ik}b_{kj} \neq c_{ij}$) **then** (**return** „$\mathbf{AB} \neq \mathbf{C}$“);
end
return „$\mathbf{AB} = \mathbf{C}$“;

Was ist die Laufzeit (Anzahl Multiplikationen) dieses Algorithmus? Sie hängt von der Stichprobengrösse ab - von der Anzahl der Matrixeinträge, die wir nachrechnen wollen. Pro Matrixeintrag müssen wir n Multiplikationen durchführen. Also benötigt der Algorithmus insgesamt $n \cdot s$ Multiplikationen.

Aufgabe 32. *Überlege dir folgende Fälle für die Stichprobengrösse s dieses Algorithmus:*

(a) Was macht der Algorithmus, wenn $s = n^2$ ist?

(b) Und was passiert, wenn man $s = 0$ wählt?

(c) Was ist die Laufzeit in den beiden Fällen (a) und (b)?

Diese beiden Fälle für s aus der letzten Aufgabe, wollen wir also sicher ausschliessen. Wir wollen s grösser als 0 und kleiner als n^2 wählen. Klar ist: Umso kleiner wir s wählen, desto besser wird die Laufzeit dieses Algorithmus. Wie wird aber die Fehlerwahrscheinlichkeit duch s beeinflusst?

Aufgabe 33. *Es gibt n^2 Matrixeinträge der Matrix $\mathbf{C}$. Der Algorithmus kontrolliert s verschiedene zufällig ausgewählte Einträge auf ihre Korrektheit.*

(a) $s = \frac{n}{2}$ *(b)* $s = n$ *(c)* $s = \frac{n^2}{2}$

Angenommen $\frac{n}{2}$ Matrixeinträge sind falsch. Wie gross ist dann die Wahrscheinlichkeit, dass der Algorithmus trotzdem “$\mathbf{AB} = \mathbf{C}$“ ausgibt? Das heisst, wie gross ist die Wahrscheinlichkeit, dass er s korrekte Matrixeinträge überprüft?

Aufgabe 34. *Betrachte noch einmal die letzte Aufgabe. Wie ändert sich diese gesuchte Wahrscheinlichkeit, wenn nur ein Matrixeintrag von* $\mathbf{C}$ *falsch ist?*

Umso grösser also die Stichprobe gewählt wird, desto sicherer wird das Resultat des Algorithmus! Das ist ziemlich klar. Wenn man mehr überprüft, wird die Wahrscheinlichket grösser, dass man einen Fehler findet. Wir konnten in den letzten beiden Aufgaben aber auch feststellen, dass die Fehlerwahrscheinlichkeit dieses Algorithmus nicht nur von s abhängt, sondern auch von der Anzahl falscher Matrixeinträge! Hat der Buchhalter bei seiner Berechnung nur genau einen Matrixeintrag falsch berechnet/oder abgeändert, so wird es viel schwieriger einen Fehler zu finden, wie wenn eine ganze Reihe von Einträgen nicht stimmen würden. Da wir aber natürlich nicht im voraus wissen, wie viele Matrixelemente nicht stimmen, oder ob überhaupt welche falsch sind, müssen wir mit der schwierigsten Situation rechnen: Genau ein Eintrag ist falsch.

Am Anfang dieses Abschnittes nahmen wir uns zum Ziel einen randomisierten Algorithmus zu erarbeiten der nur n^2 Multiplikationen durchführt und sich danach mit genügend hoher Wahrscheinlichkeit richtig entscheidet, ob er dem Buchhalter glauben soll, oder nicht. Damit unser Stichproben-Algorithmus diese gewünschte Laufzeit einhält, müssen wir s so wählen, dass:

$$\text{„Anzahl Multiplikationen des Stichpr.-Test-Alg“} = n \cdot s \leq n^2$$

gilt. Das heisst: s darf nicht grösser als n sein.

Aufgabe 35. *Angenommen nur genau ein Eintrag der* $n \times n$*-Matrix* $\mathbf{C}$ *ist falsch. Wir lassen den Algorithmus mit* $s = n$ *laufen. Wie gross ist die Fehlerwahrscheinlichkeit? Das heisst, wie gross ist die Wahrscheinlichkeit, dass der Algorithmus fälschlicherweise „*$\mathbf{AB} = \mathbf{C}$*“ ausgibt?*

Aufgabe 36. *Wendet man die in der letzten Aufgabe berechnete Formel für die Fehlerwahrscheinlichkeit dieses Algorithmus auf eine* 20×20*-Matrix an, so erhält man eine Wahrscheinlichkeit von mehr als 0.95. Wie viel mal muss man also diesen Algorithmus laufen lassen, damit man mit einer Sicherheit von 0.999 dem Reslutat „*$\mathbf{AB} = \mathbf{C}$*“ glauben kann? Und wie gross ist dann die Laufzeit des Algortihmus?*

Vergleicht man das Laufzeitsresultat der letzten Aufgabe mit der Anzahl Multiplikationen, die man durchführen müsste um alles deterministisch nachzurechnen: $20^3 = 8000$, so müssen wir leider feststellen, dass unser Stichprobenalgorithmus in diesem Fall (nur genau ein Matrixeintrag falsch) völlig

versagt! Er rechnet mehr als 6 mal länger, wie wenn er alle Einträge sauber deterministisch nachprüfen würde und hat immer noch eine Fehlerwahrscheinlichkeit von 0.001. Die Idee eine zufällige Stichprobe von nur n Einträgen als Fingerabdruck zu wählen, ist für diese schwierige Situation schlecht! Und wie man der nächsten Tabelle entnehmen kann, gilt dies nicht nur für die 20×20-Matrix:

Matrixgrösse $[n \times n]$-Matrix	Vollständiger Test [# Multiplikationen]	Stichpr.-Test-Alg. [# Multipl. damit Fehlerw. ≤ 0.001]	Stichpr.-Test-Alg. ist ...
$n = 5$	125	775	6.2 mal langsamer
$n = 10$	1'000	6'600	6.6 mal langsamer
$n = 20$	8'000	54'000	6.7 mal langsamer
$n = 50$	125'000	855'000	6.8 mal langsamer
$n = 100$	1'000'000	6'900'000	6.9 mal langsamer

Abbildung 15: Stichproben-Test-Algorithmus versagt, wenn nur ein Matrixeintrag falsch ist

Gibt es einen geschickteren Fingerabdruck? Einen Fingerabdruck der zwar auch nur n^2 Multiplikationen benötigt und trotzdem einzelne Fehler gut finden kann?

Fingerabdruck von Freivalds

Fassen wir kurz zusammen: Was wollen wir erreichen? Und wo stehen wir? Wir wollen dem Papeteriechef helfen ein gutes Verfahren zu entwicklen, das er benutzen kann, um die grosse Matrixmultiplikation $\mathbf{AB} = \mathbf{C}$ zu überprüfen. Das gewünschte Verfahren soll aber schneller sein, als das vollständige Nachrechnen aller Matrixeinträge von $\mathbf{C}$ (n^3 Multiplikationen). Wir setzten uns zum Ziel, dass der Chef nur n^2 Multipliaktionen durchführen muss für einen Test. Da man mit n^2 Multiplikationen das Matrixprodukt $\mathbf{A}\cdot\mathbf{B}$ nicht vollständig berechnen kann, können wir einen gewissen Informationsverlust nicht vermeiden. Löst man dieses Problem deterministisch, so stösst man wieder auf das Problem des Codeknackers (siehe Kapitel 2)! Der Buchhalter hat die Möglichkeit den Algorithmus im Voraus zu durchschauen, und weiss, wie er die Matrixeinträge von $\mathbf{C}$ manipulieren kann, ohne dass der Chef dies bemerken wird. Deshalb wurde unser Ziel ein randomisierter Algorithmus. Ein Algorithmus, der vom Buchhalter nicht mehr durchschaut werden kann und einer, der sich ohne vollständigen Vergleich der Matrizen $\mathbf{C}$ und $\mathbf{AB}$ möglichst korrekt entscheidet, ob „$\mathbf{AB} = \mathbf{C}$“ oder $\mathbf{AB} \neq \mathbf{C}$“ gilt. Dazu soll er von beiden Matrizen einen zufälligen Fingerabruck nehmen und die-

sen vergleichen. Hier sind wir im letzten Abschnitt stecken geblieben. Wir wählten als Fingerabruck einer Matrix eine zufällige Stichpbrobe von Matrixeinträgen. Dabei konnten wir zwar einen randomisierten Algorithmus mit der gewünschten Laufzeit n^2 Multiplikationen finden, aber die Wahrscheinlichkeit, dass der Stichproben-Test-Algorithmus einen Fehler fand, wenn es nur wenige davon gab, war zu klein! Als wir die Wahrscheinlichkeit für das Entdecken der Fehler durch Wiederholung auf 0.999 bringen wollten, mussten wir feststellen, dass man den Algorithmus so viel mal laufen lassen muss, dass er am Schluss mehr rechnet, wie, wenn man alle Matrixeinträge vollständig deterministisch überprüfen würde. Darum bleibt unser offenes Problem: Wie können wir einen Fingerabruck für eine Matrix konstruieren, der alle gewünschten 3 Bedingungen erfüllt? Ein Fingerabruck der möglichst viel relevante Information enthält, schnell berechnet werden kann und den man schnell mit anderen Fingerabdrücken vergleichen kann.

Die Lösung dazu bringt uns die Idee von Freivalds: Anstelle des Vergleichs von zufällig ausgewählten Matrixelementen, nahm Freivald einen zufälligen Vektor v, der nur die Zahlen 0 und 1 enthält, und multiplizierte diesen mit beiden Matrizen $\mathbf{C}$ und $\mathbf{AB}$. Dabei entstehen zwei Vektoren $\mathbf{C}v$ und $\mathbf{AB}v$. Diese beiden Vektoren werden verglichen. Sind sie identisch so entscheidet man sich für „$\mathbf{AB} = \mathbf{C}$“ und wenn man einen Unterschied findet gibt man „$\mathbf{AB} \neq \mathbf{C}$“ aus.

Algorithm 5: Algorithmus von Freivalds

Input: drei $n \times n$-Matrizen $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$
Output: „$\mathbf{AB} = \mathbf{C}$“ oder „$\mathbf{AB} \neq \mathbf{C}$“

$v :=$ Vektor der Grösse n, der zufällig glm. verteilt aus allen möglichen Vektoren, die nur die Zahlen 0 und 1 enthalten ausgewählt wurde;
$F_1 := \mathbf{A} \cdot (\mathbf{B} \cdot v)$; (Fingerabdruck von $\mathbf{AB}$)
$F_2 := \mathbf{C} \cdot v$; (Fingerabdruck von $\mathbf{C}$)
if *($F_1 = F_2$)* **then** (**return** „$\mathbf{AB} = \mathbf{C}$“);
if *($F_1 \neq F_2$)* **then** (**return** „$\mathbf{AB} \neq \mathbf{C}$“);

Aufgabe 37. *Im Verfahren von Freivalds muss man einen Vektor, der nur die Zahlen* 0 *und* 1 *etnhält zufällig gleichmässig verteilt aus allen möglichen solchen Vektoren auswählen.*

(a) Wie viele verschiedene Vektoren der Grösse 2 *gibt es, die nur die Zahlen* 0 *und* 1 *enthalten?*

(b) Wie gross ist die Wahrscheinlichkeit, dass der daraus zufällig ausgewählte Vektor der Grösse 2 *an der ersten Stelle eine* 1 *stehen hat?*

(c) Wie viele verschiedene Vektoren der Grösse 5 *gibt es, die nur die Zahlen* 0 *und* 1 *enthalten?*

(d) Wie gross ist bei Vektroren der Grösse 5 *die Wahrscheinlichkeit, dass an erster Stelle eine* 1 *steht?*

Betrachten wir den Algorithmus von Freivalds genauer:
Freivalds Fingerabruck einer Matrix ist das Produkt dieser Matrix mit einem Zufallsvektor v, der nur aus 0en und 1en besteht (0-1-Zufallsvektor). Schneidet dieser Fingerabdruck bei unseren erwünschten drei Bedingungen besser ab? Wie lange braucht man für die Berechnung eines Fingerabdrucks und wie aufwändig ist der Vergleich zweier Fingerabdrücke?

Aufgabe 38. *Gegeben seien folgende Produkte:*

(a) $\begin{pmatrix} 3 & 1 \\ 2 & 2 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}$

(b) $\begin{pmatrix} 2 & 0 & 1 \\ 2 & 4 & 1 \\ 3 & -1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$

Wie viele Multiplikationen müssen durchgeführt werden? (auch Multiplikationen vom Typ „$0 \cdot x$“ sollen gezählt werden)

Für die Berechnung des Fingerabdruckes müssen wir $\mathbf{C} \cdot v$ und $\mathbf{A} \cdot \mathbf{B} \cdot v$ berechnen. Um das erste Produkt zu berechnen, müssen wir den Vektor v mit jeder Zeile von $\mathbf{C}$ multiplizieren. Für jede der n Zeilen von $\mathbf{C}$ müssen wir also n Teilprodukte berechnen und addieren. So werden wir für die Berechnung von $\mathbf{C} \cdot v$ genau $n \cdot n = n^2$ Multiplikationen durchführen. Beim zweiten Produkt $\mathbf{A} \cdot \mathbf{B} \cdot v$ müssen wir etwas aufpassen. Wenn wir zuerst $\mathbf{A} \cdot \mathbf{B}$ berechnen und dann diese Produktematrix mit dem Vektor v multiplizieren, nützt uns die ganze Idee von Freivalds überhapt nichts mehr! Denn dann müssen wir die ganze Matrixmultiplikation dieser beiden $n \times n$-Matrizen $\mathbf{A}$ und $\mathbf{B}$ durchführen, was wieder der vollständigen Überprüfung aller einzelner Matrixeinträge entsprechen würde (n^3 Multiplikationen). Es ist also wichtig, dass wir zuerst die Matrix $\mathbf{B}$ mit v multiplizieren. Dies kostet uns, genau wie $\mathbf{C} \cdot v$, n^2 Multiplikationen. Dabei erhalten wir wieder einen Spaltenvektor der Grösse n und multiplizieren den noch mit der Matrix $\mathbf{A}$. So brauchen wir für die Berechnung der beiden Fingerabdrücke F_1 und F_2 drei mal n^2 Multiplikationen. Da für grosse n eine Laufzeit von $3n^2$ Multiplikationen von der selben

Grössenordung ist, wie n^2 Multiplikationen, entspricht die Berechnungszeit dieser Fingerabdrücke gerade unser Wunschvorstellung.

Auch die Zeit, die wir benötigen um zwei Fingerabrücke (zwei Spaltenvektoren der Grösse n) zu vergleichen, erfüllt die gewünschte Bedingung. Wir müssen nur n Vektoreinträge miteinander vergleichen, und überschreiten somit die Laufzeit der Grössendordnung n^2 nicht. Kommen wir wieder zum heiklen Punkt: Wie gut entdeckt dieser Algorithmus Fehler?

Aufgabe 39. *Gegeben seien folgende Matrizen:*

$$\mathbf{A} = \begin{pmatrix} 3 & 1 \\ 2 & 4 \end{pmatrix},\ \mathbf{B} = \begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix},\ \mathbf{C_1} = \begin{pmatrix} 4 & 8 \\ 6 & 10 \end{pmatrix},\ \mathbf{C_2} = \begin{pmatrix} 4 & 8 \\ 8 & 10 \end{pmatrix}$$

(a) Berechne die Matrixmultiplikation $\mathbf{A} \cdot \mathbf{B}$.

(b) Nun lässt man den Algorithmus von Freivalds zur Überprüfung von „$\mathbf{AB} = \mathbf{C_1}$*“ laufen. Bei welchen der vier möglichen Vektoren v findet dann der Algorithmus den Fehler?*

(c) Wie viele solche Vektoren sind es bei der Überprüfung von „$\mathbf{AB} = \mathbf{C_2}$*“?*

Der Fingerabdruck von Freivalds ist sehr geschickt gewählt! Wir werden nämlich zeigen können, dass der Algorithmus von Freivalds einen Fehler des Buchhalters, auch wenn es nur einen einzigen Fehler in einer sehr grossen Matrix ist, immer mit Wahrscheinlichkeit $\geq \frac{1}{2}$ finden wird. Dies ist eine enorme Verbesserung zum vorherigen Abschnitt. Die Wahrscheinlichkeit hängt bei Freivalds nicht mehr von der Anzahl Fehler, die der Buchhalter macht und von der Grösse der Matrix ab! Egal wie schwierig die Situation des Papeteriechefs ist, er wird mit dem Algorithmus von Freivalds immer in mindestens der Hälfte der Test bemerken, dass es einen Fehler gibt. Und wie wir aus Aufgabe 30 (b) wissen, reicht es, wenn man einen solchen Algorithmus 10 mal laufen lässt um eine Wahrscheinlichkeit von 0.999 zu erreichen, dass das gelieferte Resultat korrekt ist. Halten wir dieses Resulatat von Freivalds in einem Satz fest:

Satz 4. Seien $\mathbf{A}$, $\mathbf{B}$ und $\mathbf{C}$ drei $n \times n$-Matrizen.

- Gilt $\mathbf{AB} = \mathbf{C}$, so wird der Algorithmus von Freivalds immer das korrekte Resultat „$\mathbf{AB} = \mathbf{C}$“ ausgeben.
- Im Falle von $\mathbf{AB} \neq \mathbf{C}$ wird der Algorithmus von Freivalds mit einer Wahrscheinlichkeit von mindestens $\frac{1}{2}$ das korrekte Resultat „$\mathbf{AB} \neq \mathbf{C}$“ ausgeben.

Beweis von Satz 4
Der erste Punkt dieses Satzes ist klar. Wenn $\mathbf{AB} = \mathbf{C}$ gilt, muss auch für jeden beliebigen Vektor v: $\mathbf{AB}v = \mathbf{C}v$ gelten. Das heisst die Fingerabdrücke werden immer übereinstimmen und der Algorithmus wird „$\mathbf{AB} = \mathbf{C}$“ antworten.

Dass der Algorithmus aber eine falsche Rechnung des Buchhalters immer mit einer Wahrscheinlichkeit von mindestens $\frac{1}{2}$ erkennt, müssen wir uns schon etwas genauer überlegen. Wir müssen dazu zeigen, dass für mindestens die Hälfte aller möglichen Vektoren v die Fingerabrücke von $\mathbf{AB}$ und $\mathbf{C}$ verschieden sind. Wie viele solche Vektoren v gibt es insgesamt? Der Vektor v hat n Einträge $v_1, \ldots, v_n$:

$$v = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}, \text{ mit Prob}[v_i = 0] = \text{Prob}[v_i = 1] = \frac{1}{2} \text{ für } 1 \leq i \leq n$$

Jeder Eintrag v_i kann entweder 0 oder 1 sein. So gibt es also 2^n Möglichkeiten die Einträge v_1 bis v_n mit Nullen und Einsen zu füllen. Und wir wollen nun zeigen, dass es für alle beliebigen Matrizen $\mathbf{A}$, $\mathbf{B}$ und $\mathbf{C}$ mit $\mathbf{AB} \neq \mathbf{C}$ immer mindestens $\frac{2^n}{2} = 2^{n-1}$ Vektoren v gibt, die im Algorithmus von Freivalds diese Ungleichheit bezeugen; Vektoren v für die also gilt: $\mathbf{AB}v \neq \mathbf{C}v$. Wie eine normale Gleichung, können wir auch diese Ungleichheit umschreiben, indem wir auf beiden Seiten $\mathbf{C}v$ subtrahieren und danach v ausklammern:

$$\begin{aligned} \mathbf{AB}v \neq \mathbf{C}v &\iff \mathbf{AB}v - \mathbf{C}v \neq 0 \\ &\iff \underbrace{(\mathbf{AB} - \mathbf{C})}_{:=\mathbf{D}} v \neq 0 \end{aligned}$$

Die Matrix $(\mathbf{AB} - \mathbf{C})$ ist immer noch eine $n \times n$. Damit es etwas einfacher wird mit dieser Matrix weiter zu überlegen, nennen wir sie im folgenden nicht mehr $(\mathbf{AB} - \mathbf{C})$, sondern einfach $\mathbf{D}$. Da wir davon ausgehen, dass $\mathbf{AB} \neq \mathbf{C}$ gilt, muss mindestens ein Eintrag von $\mathbf{AB}$ ungleich dem entsprechenden Eintrag von $\mathbf{C}$ sein. Das heisst, es muss mindestens einen Eintrag in $\mathbf{D}$ geben, der nicht 0 ist.

Wir betrachten die Zeile in $\mathbf{D}$, in der sich dieser Eintrag d_{ij}, der sicher nicht 0 ist, befindet:

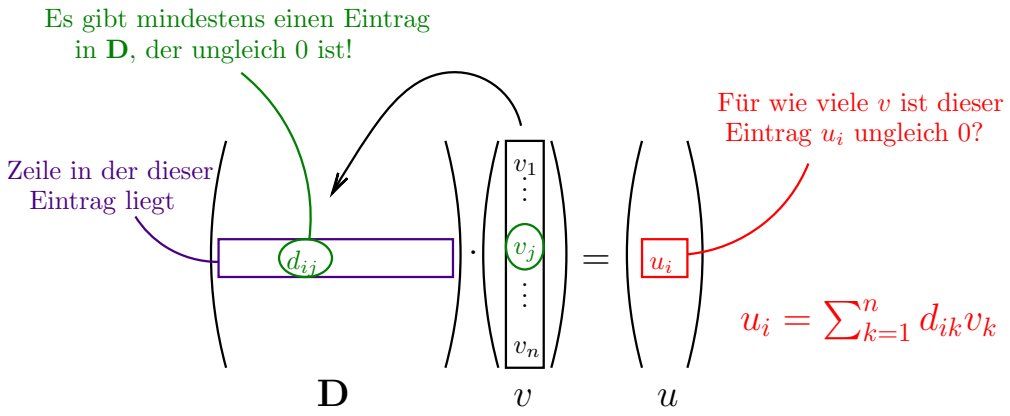


Abbildung 16: Alle v für die $\mathbf{D}v \neq 0$ ist können die Ungleichheit bezeugen!

Wie viele v gibt es mindestens, für die $\mathbf{D}v \neq 0$ ist? Wir wollen zeigen, dass es bereits 2^{n-1} Vekroren v gibt, für die der Eintrag u_i (siehe Abbildung 16) des Produktevektors $\mathbf{D}v = u$ ungleich 0 ist. Das machen wir wie folgt: Um den Eintrag u_i zu berechnen, müssen wir den Vektor v mit der i-ten Zeile von $\mathbf{D}$ multiplizieren, d.h.:

$$\begin{aligned} u_i &= v_1 \cdot d_{i1} + \ldots + v_{j-1} \cdot d_{ij-1} + v_j \cdot \underbrace{d_{ij}}_{\neq 0} + v_{j+1} \cdot d_{ij+1} + \ldots + v_n \cdot d_{in} \\ &= \underbrace{(v_1 \cdot d_{i1} + \ldots + v_{j-1} \cdot d_{ij-1} + v_{j+1} \cdot d_{ij+1} + \ldots + v_n \cdot d_{in})}_{\text{ergibt aufaddiert irgend eine Zahl, nennen wir sie := x}} + (v_j \cdot \underbrace{d_{ij}}_{\neq 0}) \end{aligned}$$

Wie müssen wir jetzt $v_1, \ldots, v_n$ wählen, damit $u_i \neq 0$ ist? Wir wählen für alle Einträge von v ausser für v_j (jener Eintrag von v, der mit unserem d_{ij} multipliziert wird, von dem wir wissen, dass $d_{ij} \neq 0$ ist) einen beliebigen Eintrag 0 oder 1. Dann gilt:

$$u_i = x + d_{ij} \cdot v_j$$

So ist der Eintrag u_i bis auf die Wahl von v_j bestimmt. Die Variable x ist durch die Einträge der Zeile von $\mathbf{D}$ und von den $n-1$ beliebig gewählten Einträgen von v bestimmt. Wie gross x ist wissen wir nicht! Es kann 0 sein, eine beliebige negative Zahl oder natürlich auch eine beliebige positive Zahl. Das Wichitge ist jetzt aber, dass man v_j so wählen kann, dass $u_i \neq 0$ ist. Für die Wahl von v_j haben wir nämlich zwei Varianten: Entweder wählen wir $v_j = 0$ oder wir wählen $v_j = 1$. Das heisst wir können zwischen zwei Varianten für u_i auswählen:

1.Variante: $u_i = x + d_{ij} \cdot 1 = x + d_{ij}$

2.Variante: $u_i = x + d_{ij} \cdot 0 = x$

Da wir wissen, dass $d_{ij} \neq 0$ ist, sind das auch wirklich zwei verschiedene mögliche Wahlen für u_i, die uns zur Verfügung stehen. Das heisst eine dieser zwei unterschiedlichen Zahlen (x oder $x + d_{ij}$) muss sicher ungleich 0 sein! Es ist also möglich alle Einträge von v bis auf diesen einen speziellen Eintrag v_j beliebig zu wählen. Anschliessend ist es dann immer noch möglich durch die Wahl des letzten Eintrags v_j von v das Ergebnis so zu bestimmen, dass $\mathbf{D}v$ sicher ungleich 0 ist. Für die beliebige Wahl von $n-1$ Einträgen von v (jeweils 0 oder 1) gibt es 2^{n-1} Möglichkeiten!

Somit haben wir gezeigt, dass es für jede beliebige Matrix $\mathbf{D} \neq \mathbf{0}$ sicher 2^{n-1} mögliche Vektoren v gibt, die im Algorithmus von Freivalds bezeugen werden, dass diese Matrix ungleich null ist (oder eben analog, dass $\mathbf{AB} \neq \mathbf{C}$ gilt). Und, dass deshalb die Wahrscheinlichkeit, dass der Algorithmus bei einer zufällig gleichmässigen Wahl eines Vektors v aus allen möglichen 2^n Vektoren einen der mindestens 2^{n-1} günstigen Vektoren erwischt, $\geq \frac{1}{2}$ ist. □

Wir haben also einen Fingerabdruck gefunden, der sowohl die Laufzeit (Anzahl Multplikationen) einhält, als auch mit genügend hoher Wahrscheinlichkeit die Fehler des Buchhalters entdecken wird. Schauen wir uns das Resultat in der folgenden Tabelle an:

Matrixgrösse $[n \times n]$-Matrix	Vollständiger Test [# Multiplikationen]	Alg. v. Freivalds [# Multipl. damit Fehlerw. ≤ 0.001]	Alg. v. Freivalds ist ...
$n = 5$	125	250	2 mal langsamer
$n = 10$	1'000	1'000	gleich schnell
$n = 20$	8'000	4'000	2 mal schneller
$n = 50$	125'000	25'000	5 mal schneller
$n = 100$	1'000'000	100'000	10 mal schneller
$n = 1000$	1'000'000'000	10'000'000	100 mal schneller

Abbildung 17: Vorteil für grosse Matrizen

Für kleine Matrizen bis zu 10×10-Matrizen bringt uns der Algorithmus von Freivalds noch keine Vorteile. Dann lohnt sich das vollstädnge Durchtesten aller n^2 Einträge der Matrix $\mathbf{C}$ mehr. Wenn hingegen die Matrizen viel grösser werden, so wie es bei unserem Papeteriechef der Fall ist, so wird das

randomisierte Verfahren von Freivalds immer besser!

Wir haben in diesem Kapitel also einen weiteren Vorteil von randomisierten Algorithmen kennengelernt. Randomisierte Algorithmen können im Vergleich zu deterministischen Algorithmen nicht nur die Durchschaubarkeit des Verfahrens gegenüber von Gegnern verhindern (Codeknacker), sondern auch Verfahren verkürzen, indem man eine kleine Fehlertoleranz zulässt!