Pascalsches Dreieck

Finn Jannsen, Philipp Schwarz

22. April 2019

Inhaltsverzeichnis

1 Einführung

Diese Dokumentation beschreibt drei Implementations-Varianten zur Berechnung einer Reihe aus dem Pascalschen Dreieck. Hierbei wurde eine Rekursive, eine Iterative und eine Strategie, die mittels Binominalkoeffizient berechnet angewandt. In Abschnitt 2 wird darauf eingegangen, wie die verschiedenen Varianten realisiert wurden. Anschließend wird in Abschnitt 3.1 geprüft, ob die Varianten korrekt funktionieren und in Abschnitt 3.2 die Performance der Varianten verglichen.

2 Implementation

Die drei Varianten wurden in Java als Klassen implementiert. Da alle den gleichen Funktionssatz brauchen implementieren sie alle das gleiche Interface namens PascalTriangleCalculator.
Da die Klassenstrukturen im wesentlichen nur eine Methode beinhalten (calculateRow(int rowNum)), die die gewünschte Reihe berechnet und als Integer-Array zurückgibt, liegt im Folgenden der Fokus auf der Implementierung der Algorithmen. Im Folgenden gilt: n bezeichnet die Reihe und k die Stelle in der Reihe

$$(n, k) = (n - 1, k - 1) + (n - 1, k)$$

Abbildung 2.1: Berechnung von (n,k) anhand benachbarter Werte

```
if (rowNum > 1) {
    pre = calculateRow(rowNum - 1);
} else if (rowNum == 1) {
    // special case row n=1 only contains 1
    pre = new int[]{1};
}

int[] row = new int[rowNum];

for (int i = 0; i < rowNum; i++) {
    if (i == 0 || i == rowNum - 1) {
        // special case on row boundaries
        row[i] = pre[0];
    } else if (i == 1 || i == rowNum -2) {
        // special case on row boundaries neighbours
        row[i] = rowNum - 1;
    } else {
        row[i] = pre[i-1] + pre[i];
    }
    counter++;
}
return row;
```

Abbildung 2.2: Code-Ausschnitt rekursive Strategie

2.1 Rekursive Strategie

Die rekursive Strategie wird umgesetzt mit einem rekursiven Aufruf. Um eine Reihe zu berechnen wird die Methode mit dem Argument der Reihe n-1 aufgerufen und danach mithilfe der Formel aus 2.1 die einzelnen Werte der Reihe berechnet, welche als Rückgabewert an die aufrufende Funktion weitergegeben werden.
In Abbildung 2.2 ist dies ersichtlich.

2.2 Iterative Strategie

In der iterativen Strategie wird lediglich von Reihe 1 an bis zur gewünschten Reihe mittels der Formel 2.1 aufaddiert.

2.3 Binominalkoeffizient Strategie

Da das Pascalsche Dreieck die Binominalkoeffizienten repräsentiert lassen sich die einzelnen Werte auch wie folgt errechnen:

$$\binom{n}{k}$$

Dies wurde in Java durch die normale Berechnung eines Binominalkoeffizienten umgesetzt und wird im folgenden mit den anderen Lösungen verglichen.

3 Verifizieren und Testen

3.1 Verifizieren der Funktionalität

Alle Strategien wurden auf ihre Funktionalität durch einen Vergleich von ein paar resultierenden Reihen und von Hand errechneten Ergebnissen überprüft. Alle Tests wurden erfolgreich absolviert.

3.2 Aufwandsanalyse

Für die Aufwandsanalyse wurde zu jedem Algorithmus ein Counter eingeführt, der die reinen Rechenoperationen zählt. Das heißt, dass der Counter bei der iterativen und rekursiven Strategie bei den Additionen, bei jeder Multiplikation der Binominalkoeffizient-Strategie und generell bei der Annahme vom Wert=1 an Stelle $k = 1$ und $k = n$ oder deren Nachbarn mit dem Wert=$n - 1$ hochzählt. Die Beobachtungen sind in Abbildung 3.1 zu sehen.

Auffällig ist hier erstmal, dass alle Methoden in doppelt-Logarithmischer Darstellung einer Ursprungsgerade ähneln. Dies sagt aus, dass die Anzahl der Rechenoperationen einer Potenzfunktion gleicht. Auch fällt auf, dass sich iterative und rekursive Strategie gleich verhalten. Das liegt daran, dass sie auf die gleiche Weise funktionieren, jedoch unterschiedliche Herangehensweisen haben. Die Linie der Rechenoperationen beim Binominalkoeffizienten liegt parallel über den anderen, was heißt dass sie sich bei Werten $\lim_{n\to\infty}$ nur durch einen konstante Faktoren unterscheiden. Um den Asymptotischen Aufwand zu untersuchen wurden die Ergebnisse bei höheren Reihen (z.b. $n = \{2048, 4096, 8192\}$) durch n geteilt. Daraufhin wiesen die Ergebnisse auf, dass sie immer knapp unter n (Bino.) und $\frac{n}{2}$ (Iter./Rek.) lagen. Also wurden die Ergebnisse mit $f(n) = n^2$ und $g(n) = \frac{n^2}{2}$ verglichen und die obige Prozedur mit der Differenz zu den echten Werten wiederholt. Durch annähern wurden also folgende Formeln zur Begrenzung nach oben herausgearbeitet:

Rechenoperationen bei iterativer und rekursiver Strategie:

$$f(n) = \frac{n^2}{2} + \frac{n}{2}$$

Rechenoperationen bei Binominalkoeffizient-Strategie:

$$g(n) = n^2 + 4n$$

Mit diesen Formeln ergibt sich nun der Asymptotische Aufwand, indem man die konstanten Faktoren entfernt:

$$\Theta(f(n)) = \Theta(g(n)) = \Theta(n^2)$$

Es befinden sich also alle drei Methoden in der gleichen Asymptotischen Klasse.

Weiter untersucht wurde der echte Aufwand auch durch die Laufzeit in ms, zu sehen in Abbildung 3.2 (hier lineare Darstellung). Grundsätzlich haben hier die rekursive Strategie und der Binominalkoeffizient etwa die gleiche Laufzeit. Im Vergleich dazu dauert die iterative Strategie etwas länger, was evtl. eine hohe Nutzung der Resourcen vom Algorithmus zur Grundlage hat. Es lässt sich anmerken, dass die rekursive Strategie bei hohen Reihen eine Veränderung der Stackgröße der Java VM benötigt, wegen der vielen Methodenaufrufe. Was sich aus diesem Diagramm auch ablesen lässt ist, dass die hohe Anzahl an Rechenoperationen des Binominalkoeffizienten keinen merkbaren Einfluss hat. Dies liegt an der Effizienz von Multiplikation auf Hardware-Ebene.

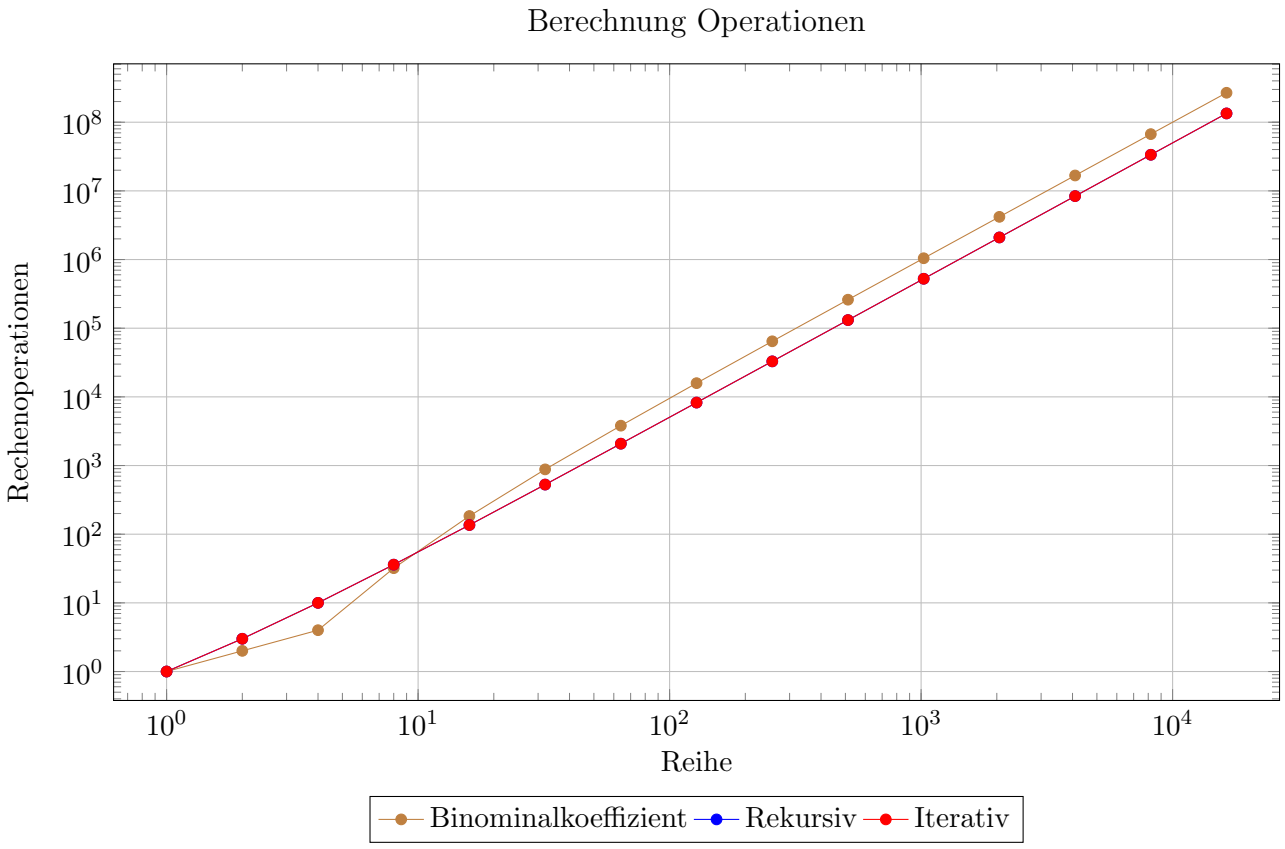

Abbildung 3.1: Quantitativer Vergleich der Berechnung einer Reihe im Pascalschen Dreieck in loglog-Darstellung

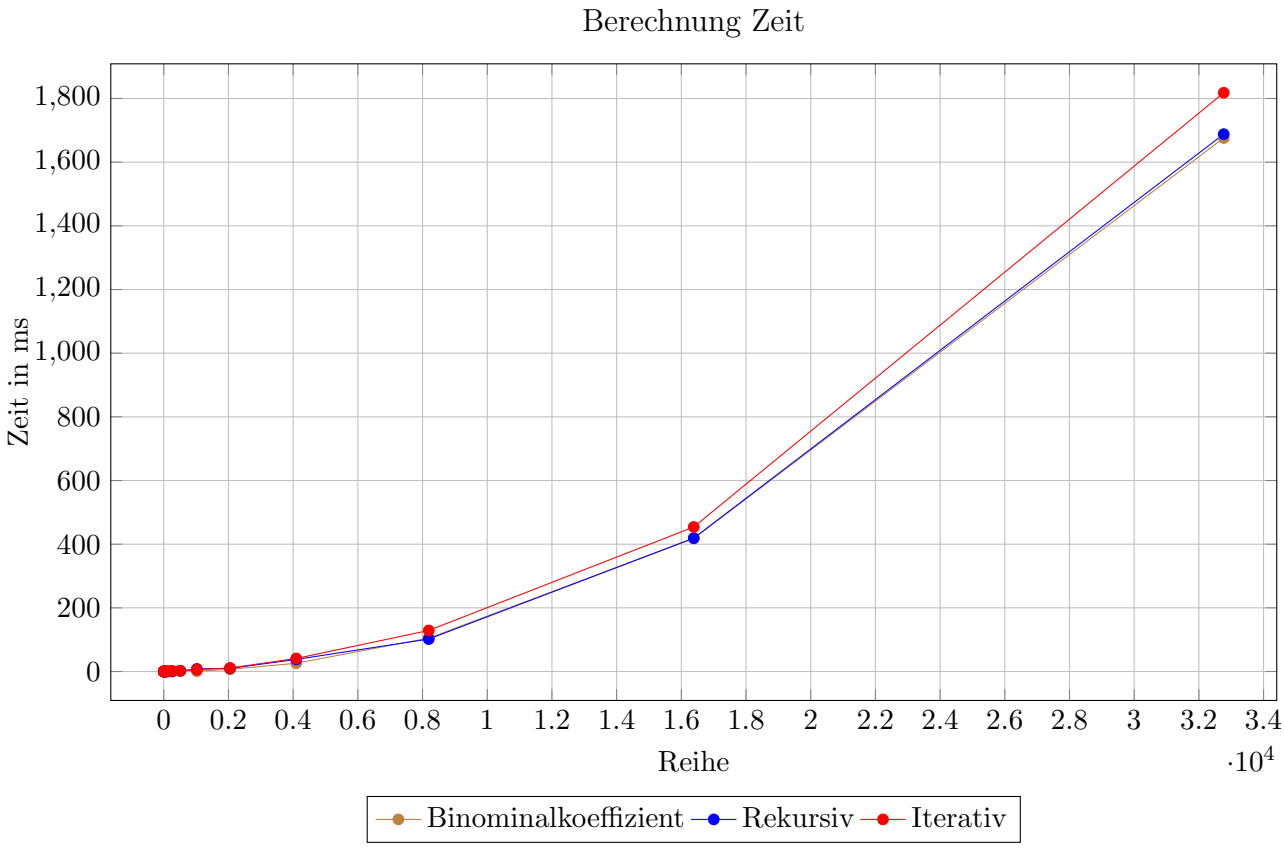

Abbildung 3.2: Quantitativer Zeitvergleich der Berechnung einer Reihe im Pascalschen Dreieck