Potenzen algorithmisch berechnen – Ein einführendes Beispiel in die Komplexitätstheorie

Simon Knellwolf

Diese Unterrichtseinheit entstand im Rahmen des CAS-Basiskurses für Junglehrkräfte im April 2008 in Zürich unter der Leitung von René Hugelshofer.

Stufe: Angewandte Mathematik in der Sekundarstufe II

Dauer: eine Doppelstunde

Voraussetzungen: Potenzgesetze, elementare Programmierkenntnisse (Kontrollstrukturen, Funktionen)

Unter der Komplexität eines Algorithmus versteht man meist den Verbrauch an Speicherplatz oder Rechenzeit, den er zur Lösung eines bestimmten Problems benötigt. In dieser Unterrichtseinheit geht es nur um letztere. Wir implementieren einen einfachen - aber langsamen - Algorithmus zur Berechnung der n-ten Potenz einer natürlichen Zahl und versuchen ihn zu verbessern.

Der TI-Nspire eignet sich hervorragend für den praktischen Teil dieser Unterrichtseinheit, da sein Prozessor im Vergleich zu Prozessoren von Einzelplatzrechnern langsam arbeitet. Dadurch wird jede Effizienzsteigerung des Algorithmus bereits beim einfach verständlichen Problem der Potenzberechnung erfahrbar. Diese Unterrichtseinheit will einen intuitiven, experimentellen Zugang zur Effizienz von Algorithmen schaffen.

Das Aufgabenblatt bildet den Kern der Unterrichtseinheit. Es gewährleistet einen hohen Anteil an Eigenaktivität der Lernenden. Die Lehrperson soll – falls nötig – geeignete Hilfestellungen für das Programmieren von eigenen Funktionen im TI-Nspire anbieten.

Aufgabenblatt: Potenzen algorithmisch berechnen

Aufgabe 1: Algorithmus zur Potenzberechnung

In dieser Aufgabe sollst du einen ersten Algorithmus zur Berechnung der n-ten Potenz einer natürlichen Zahl a programmieren.

a) Berechne ohne Taschenrechner die Potenzen 5^3, 3^4 und 2^7.
b) Wieviele Multiplikationen benötigst du zur Berechnung der obigen Potenzen? Wie viele benötigst du für a^n?
c) Beschreibe dein Vorgehen zur Berechnung von a^n in Worten.
d) Programmiere das beschriebene Vorgehen mit Hilfe einer For-Schleife als Funktion `potenzschnecke(a,n)`.
e) Führe `potenzschnecke(2,2100)` aus und vergleiche dessen Rechenzeit mit jener der Tastenfolge `2^2100`.

Aufgabe 2: Algorithmus verbessern

Offensichtlich berechnet der TI-Nspire a^n auf eine andere (schnellere) Art als wir das in `potenzschnecke(a,n)` machen. In dieser Aufgabe sollst du `potenzschnecke(a,n)` zu `potenzigel(a,n)` verbessern, welche auch Potenzen mit bestimmten grossen Exponenten ähnlich schnell berechnet wie die Tastenfolge `a^n`.

a) Mit Hilfe der Potenzgesetze lässt sich 3^8 in nur drei Multiplikationsschritten berechnen. Erkläre.
b) Wieviele Multiplikationen benötigst du für 3^{16}, 3^{64} und 3^{1024}?
c) Wie könntest du vorgehen, um 3^{17} mit weniger als 16 Multiplikationen zu berechnen?
d) Beschreibe dein verbessertes Vorgehen zur Berechnung von a^n in Worten.
e) Programmiere das beschriebene Vorgehen als Funktion `potenzigel(a,n)`.
f) Führe `potenzigel(2,2100)` aus und vergleiche mit `potenzschnecke(2,2100)` und der Tastenfolge `2^2100`.

Aufgabe 3: Algorithmus nochmals verbessern

Die Funktion `potenzigel` berechnet 2^{2100} deutlich schneller als die Funktion `potenzschnecke`. Sie entspricht aber noch immer nicht der Methode, nach welcher der TI-Nspire Potenzen berechnet. Dies wird deutlich, wenn du die Rechenzeiten für `potenzigel(2,2000)` und für die Tastenfolge `2^2000` vergleichst.

a) Vergleiche `potenzigel(2,2100)` mit `potenzigel(2,2000)`. Erkläre.
b) Welche Ideen hast du für `potenzhase(a,n)`? Formuliere sie in Worten und versuche sie zu implementieren.

Lösungen und Hinweise zum Aufgabenblatt

Aufgabe 1

a) $5^3 = 125$, $3^4 = 81$, $2^7 = 128$. Die meisten Lernenden werden die Potenzen berechnen, indem sie den Exponenten schrittweise um 1 erhöhen. Falls jemand bereits die Idee $3^2 \cdot 3^2 = 3^4$ hat, soll er Aufgabe 1 trotzdem mit Hilfe von Einerschritten beim Exponenten lösen, damit er `potenzschnecke` und `potenzigel` vergleichen kann.

b) Für a^n werden $n - 1$ Multiplikationen benötigt.

c) Falls $n > 1$ multipliziere man a mit a, falls $n > 2$ multipliziere man das Produkt nochmals mit a, falls $n > 3$ multipliziere man das neue Produkt nochmals mit a, usw. Nach $n-1$ Multiplikationen entspricht das Produkt der Potenz a^n.

d)

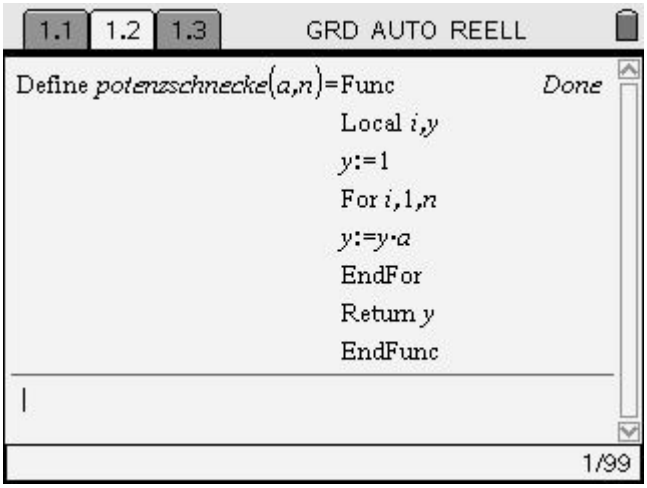

Abb. 1

e) Die Funktion `potenzschnecke` benötigt offensichtlich viel mehr Rechenzeit als die implementierte Funktion des TI-Nspire. Wir haben folglich noch nicht den bestmöglichen Algorithmus zur Lösung unseres Problems gefunden.

Hinweis: Der TI-Nspire kann Zahlen bis 10^{999} handhaben. Theoretisch erlaubt dies Zweierpotenzen mit Exponenten $n < 3321$.

Aufgabe 2

a) Man berechnet $(((3^2)^2)^2$.

b) Man benötigt 4, 6 respektive 10 Multiplikationen.

c) Man könnte $((((3^2)^2)^2)^2 \cdot 3$ berechnen. Das sind 5 Multiplikationen.

d) Anstatt den Exponenten in Einerschritten zu erhöhen, verdoppeln wir ihn schrittweise wie in den obigen Beispielen bis wir den gewünschten Exponenten überschreiten würden. Nur die fehlenden Erhöhungen machen wir in Einerschritten.

e)

Abb. 2

f) Bei den Laufzeiten von `potenzigel(2,2100)` und der Tastenfolge `2^2100` ist kein Unterschied feststellbar. Es besteht wohl trotzdem ein kleiner Unterschied. Unser Algorithmus benötigt 63 Multiplikationen (11 bis 2^{2048}, anschliessend noch 52), der Taschenrechner verwendet wahrscheinlich den Algorithmus Square and Multiply (siehe Wikipedia) und benötigt damit sicher weniger als $2[\log_2(2100) + 1] = 24$ Multiplikationen.

Aufgabe 3

a) Die Funktion `potenzigel` berechnet 2^{2100} deutlich schneller als 2^{2000}. Das wird klar, wenn wir die Anzahl Multiplikationen in beiden Fällen ausrechnen: für 2^{2100} benötigt `potenzigel` 63 Multiplikationen, für 2^{2000} aber deren 986 (10 bis 2^{1024}, anschliessend noch 976).

b) Es liegt auf der Hand, die Zwischenresultate zu speichern und wiederzuverwenden. So könnte beispielsweise 2^{10} als $((2^2)^2)^2 \cdot 2^2$ berechnet werden. Dieser Ansatz hat einen steigenden Speicherbedarf zum Nachteil. Der Ausgang dieser Aufgabe ist sehr offen. Interessierte Lernende sollen auf Square and Multiply verwiesen werden.

Literatur

[1] Für den Laien verständliche, ausführlichere Hinweise zur Komplexitätstheorie: Armin Barth, *Algorithmik für Einsteiger*, Vieweg, 2003, Braunschweig.

[2] Einführung zum Programmieren mit TI-Nspire CAS: Josef Böhm, *Programmieren mit dem TI-Nspire CAS*, bk teachware, 2008, Linz.

Simon Knellwolf, simon.knellwolf@sunrise.ch, Brugg AG, Kantonales Gymnasium Menzingen