Grenzen der Berechenbarkeit (Fortsetzung)

Laufzeitordnungen

Wir haben gesagt, dass die Laufzeit des Kennwortknackerprogramms – unabhängig davon, wie sich verschiedene Parameter im Detail auswirken – mit zunehmender Zahl verwendeter Zeichen und steigender Anzahl an Stellen des Kennworts exponentiell wächst. Deshalb sprechen wir von einem **exponentiellen Laufzeitverhalten** und ordnen den Algorithmus in eine **Laufzeitordnung** ein, die wir **exp(n)** nennen.

Hintergrundinformation: Schön und gut, dass wir bei der Einordnung in eine Laufzeitordnung Summanden und Faktoren in unserer Formel einfach ‚weg-abstrahieren'. Aber müssen wir nicht dennoch unterscheiden, ob Variablen in der Basis oder im Exponenten stehen? Die Argumentation, warum das keinen Unterschied macht, lautet wie folgt:
Wie wir wissen, ist ln(x) diejenige Zahl, mit der wir die Eulersche Zahl e potenzieren müssen, um x zu erhalten: $e^{\ln(x)} = x$. Außerdem wissen wir (möglicherweise), dass $\ln(x^y) = y \cdot \ln(x)$ – wer's nicht glaubt, probiert es an ein paar Beispielen mit dem Zehnerlogarithmus aus.
Demnach gilt dann: $x^y = e^{\ln(x^y)} = e^{y \cdot \ln(x)}$. Somit ist die Unterscheidung zwischen Basis und Exponent (für unsere Zwecke) hinfällig: Beides lässt sich als Exponent ausdrücken.

Nun wollen wir weitere Laufzeitordnungen kennenlernen. Zwei ganz grundsätzlich andere können wir erschließen:

1. **Laufzeitordnung** ____, _____ **Laufzeitverhalten**:

Aufgabe zu 1: Nehmen wir an, wir haben ein String-Array der Größe n erzeugt `(val meinArray = Array(n){_ -> “”})` und es mit Daten gefüllt, ohne diese Daten irgendwie zu sortieren. Nun wollen wir herausfinden, ob eine bestimmte Zeichenkette in dem Array abgespeichert ist. Schreiben Sie für eine Klasse mit dem Attribut `meinArray` eine Methode `suchen(suchbegriff: String): Boolean`, die das leistet.

Klären Sie, wie sich die Laufzeit Ihrer Methode (worst case) verändert, wenn sich die Arraygröße n und damit die Anzahl der (möglicherweise) abgespeicherten Zeichenketten ändert.

2. **Laufzeitordnung** _____, _____ **Laufzeitverhalten**:

Aufgabe zu 2: Nun sollen die Daten so gespeichert werden, dass ein gesuchtes Objekt schneller gefunden werden kann. Das ist zum Beispiel in einem Binärbaum der Fall, in dem die Objekte sortiert gespeichert werden. Das Lernvideo unter https://studyflix.de/informatik/binaerer-suchbaum-1364 zeigt (einfach am Beispiel von Zahlen), wie das funktioniert. Sehen Sie sich das Video bis Minute 2:25 an. Erschließen Sie dann das Laufzeitverhalten der – im Video gezeigten – Suche in einem Baum: Dazu müssen Sie sich vergegenwärtigen, dass in einem Binärbaum mit einer ‚Ebene' ein Wert gespeichert werden kann, in einen Binärbaum mit zwei ‚Ebenen' passen drei Werte, in einen Binärbaum mit drei ‚Ebenen' sieben Werte, in einen Binärbaum mit x ‚Ebenen' 2^x-1 Werte.

Weitere, enger gefasste, Laufzeitordnungen, die im Zusammenhang mit der Optimierung von Laufzeiten von Interesse sind und deshalb nicht unter einer möglichen Oberkategorie belassen werden, sind die **Laufzeitordnung n^2** und die **Laufzeitordnung $n \cdot \log(n)$**.

Versuchen Sie nachzuvollziehen, warum diese bei den folgenden Sortieralgorithmen – also Algorithmen zur Sortierung von Inhalten, die z. B. in einem Array gespeichert sind – vorliegen:
‚Insertion Sort', wie er unter https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html *vorgeführt wird, und*
‚Mergesort', wie in der Abbildung unter https://medium.com/javarevisited/visualizing-designing-and-analyzing-the-merge-sort-algorithm-904ceb78a592 *veranschaulicht ist.*