Zeitkomplexitäten in Groß-O-Notation

Was bedeuten bestimmte Zeitkomplexitäten anschaulich?

Die folgende Tabelle stammt aus http://de.wikipedia.org/wiki/Landau-Symbole#Beispiele_und_Notation und wurde nur geringfügig geändert und angepasst.

Sei `f(n)` eine (Schrittzahl-) Funktion mit einem positiv-ganzzahligen Parameter `n` (der die Problemgröße darstellt).

Notation	**Bedeutung**	**Anschauliche Erklärung**	**Ein Beispiel-Algorithmus mit dieser Zeikomplexität**
$f \in O(1)$	`f` ist beschränkt	Alle Werte von `f` liegen (unabhängig von `n`) unterhalb einer Konstanten.	Zugriff auf die `n`-te Komponente `r[n]` einer Reihung `r`.
$f \in O(\log_2(n))$	`f` wächst logarithmisch	`f` wächst ungefähr um 1 wenn man `n` ver-2-facht .	Binäre Suche in einer sortierten Reihung der Länge `n`.
$f \in O(\sqrt{n})$	`f` wächst wie die Quadratwurzel-funktion	`f` wächst ungefähr auf das 2-fache, wenn man `n` ver-4-facht.	Naiver Primzahltest mittels Teilen durch jede ganze Zahl $\leq\sqrt{n}$.
$f \in O(n)$	`f` wächst linear	`f` wächst ungefähr auf das 2-fache, wenn man `n` ver-2-facht.	Suche in unsortierter Reihung der Länge `n`
$f \in O(n * \log(n))$	`f` wächst super-linear		Sehr gute Algorithmen zum Sortieren einer Reihung der Länge `n`, z.B. Mergesort, Heapsort
$f \in O(n^2)$	`f` wächst quadratisch	`f` wächst ungefähr auf das 4-fache, wenn man `n` ver-2-facht.	Einfache Algorithmen zum Sortieren einer Reihung der Länge n, z.B. Selectionsort.
$f \in O(2^n)$	`f` wächst exponentiell	`f` wächst ungefähr auf das 2-fache, wenn man `n` um 1 erhöht.	Erfüllbarkeitsproblem der Aussagenlogik (SAT) mittels exhaustivem Verfahren
$f \in O(n!)$	`f` wächst faktoriell ("wie die Fakultäts-funktion")	`f` wächst ungefähr auf das (n+1)-fache, wenn man `n` um 1 erhöht.	Problem des Handlungsreisenden

Erweitern Sie diese Tabelle um ein paar Zeilen, indem Sie die folgenden Aufgaben lösen.

Aufgabe-01: Füllen Sie die leeren Felder aus:

Angenommen die Schrittzahl-Funktion f(n) wächst wie	**Das bedeutet: Die Schrittzahl f(n) wächst ungefähr auf das 2-fache wenn man die Problemgröße n um folgenden Faktor vergrößert:**
die 2-te Wurzel von n (Quadratwurzel)	
die 3-te Wurzel von n (Kubikwurzel)	
die 4-te Wurzel von n	
die m-te Wurzel von n	

Aufgabe-02: Füllen Sie die leeren Felder aus:

Angenommen die Schrittzahl-Funktion f(n) wächst wie	**Das bedeutet: Wenn man die Problemgröße n ver-2-facht, dann wächst die Schrittzahl f(n) ungefähr um den folgenden Faktor:**
die 2-te Potenz von n (d.h. wie n^2)	
die 3-te Potenz von n (d.h. wie n^3)	
die 4-te Potenz von n (d.h. wie n^4)	
die m-te Potenz von n (d.h. wie n^m)	

Wozu werden Zeitkomplexitäten in der Praxis benutzt?

Angenommen, für ein algorithmisches Problem gibt es mehrere Lösungen mit unterschiedlichen Zeitkomplexitäten. Wenn man dann eine dieser Lösungen konkret implementiert (z.B. in Java), dann sollte man in der Dokumentation der betreffenden Klasse(n) die Zeitkomplexität der Lösung angeben.

Andersherum: Wenn man z.B. eine Sammlungsklasse aus der Java-Standardbibliothek benutzen will (z.B. die Klasse `ArrayList` oder die Klasse `LinkedList`), sollte man sich vorher die Zeitkomplexitäten der wichtigsten Methoden (zum Einfügen, Suchen und Entfernen von Komponenten) in der Dokumentation ansehen, um die richtige Klasse auszuwählen.

Es folgen hier ein paar Auszüge aus der Online-Dokumentation der Java-Standardklassen (Java Platform Standard Ed. 8), Hervorhebungen durch kursive Schrift wurden hier hinzugefügt:

Aus der Dokumentation der Klasse **ArrayList<K>**:

The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The `add` operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. All of the other operations run *in linear time* (roughly speaking). The constant factor is low compared to that for the `LinkedList` implementation.

Aus der Dokumentation der Klasse **LinkedList<K>**:

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

Aus der Dokumantation der Klasse **TreeSet<K>**:

This implementation provides guaranteed *log(n) time* cost for the basic operations (`add`, `remove` and `contains`).

Aus der Dokumentation der Klasse **HashSet<K>**:

This class offers *constant time* performance for the basic operations (`add`, `remove`, `contains` and `size`), assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires time proportional to the sum of the `HashSet` instance's size (the number of elements) plus the "capacity" of the backing `HashMap` instance (the number of buckets). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.