Laufzeitkomplexität von Algorithmen[1]

Die Untersuchung der Effizienz von Algorithmen ist eine klassische und zugleich auch wichtige Aufgabe der Informatik, sind schnelle und stabile Algorithmen doch ein Muss jedes größeren Programms.

Dementsprechend muss es auch Verfahren geben, mit denen die Laufzeit eines Algorithmus' untersucht werden kann. Dieses wollen wir am Beispiel der Laufzeitkomplexität für Selection[2]- und Bubble-Sort erarbeiten: Wir hatten schon erkannt, dass das bloße Zählen der Kartenvertauschungen nicht sonderlich aussagekräftig ist. Vielmehr scheint es sinnvoll zu sein, die Zugriffe auf die zu sortierenden Elemente zu zählen. Daher vereinbaren wir:

Zählweise:
- Das Vergleichen zweier Zahlen aus dem Array kostet zwei Zugriffe
- Das Vertauschen zweier Zahlen im Array kostet sechs Zugriffe
- Zugriffe auf Hilsvariablen (zum Beispiel den Schleifenzähler) zählen wir nicht mit, da uns nur die sogenannten *Elementaroperationen*[3] interessieren.

Aufgabe:

a) Nutze die vorgestellten Programme, um mit Hilfe von ①Selection-Sort und ②Bubble-Sort und nach unserer vereinbarten Zählweise die Komplexität festzustellen.

Führe einen (bzw. mehrere) Versuch(e) – für unterschiedliche Problemgrößen - durch und ergänze damit die unten aufgeführte Tabelle.

Problem-größe n	**Selectionsort**			**Bubblesort**									
	Mein Versuch	Kurs-durch-schnitt	*Operat.* / *n*	mein Versuch	Kurs-durch-schnitt	*Operat.* / *n*	1. Versuch	2. Versuch	3. Versuch	4. Versuch	5. Versuch	Durch-schnitt	*Operat.* / *n*
20													
40													
60													
80													
100													
200													
300													
400													
500													

b) Ermittle mit Hilfe eines Tabellenkalkulationsprogramms den Graphen der Zuordnung.

---

1 Aus: (c) 2004 by RTC, www.linux-related.de (Dieses Dokument unterliegt der GNU Free Documentation License) und Jens Gallenbacher: Abenteuer Informatik – Spektrum Akademischer verlag, S. 59ff

2 Alternative Bezeichnungen des Algorithmus sind MinSort (von Minimum) bzw. MaxSort (von Maximum), Selectsort oder ExchangeSort (AustauschSort). [Quelle: http://de.wikipedia.org/wiki/Selectionsort, 8.09.21]

3 Das sind die Operationen, die unmittelbar mit der eigentlichen Aufgabe – hier also mit dem Sortieren – nichts zu tun haben.

Selbstverständlich ist das Ergebnis davon abhängig, wie gut die Zahlen jeweils gemischt waren. Wenn man zufällig eine bereits sortierte Folge als Ausgangssituation hat, dann benötigt man zum Beispiel gar keine Vertauschungen. Bis auf solche Spezialfälle können wir an der Tabelle also recht gut ablesen, wie sich die Rechenzeit in Abhängigkeit der Problemgröße entwickelt.

Es war sinnvoll, auch die Anzahl Zugriffe pro Karte, also die Anzahl der Elementaroperationen in Abhängigkeit von der Problemgröße, zu protokollieren.

Wie kann man diesen Quotienten interpretieren? Genau wie beim komplett manuellen Sortieren, steigt der Aufwand pro Karte mit der Anzahl der Karten.

Man sagt, die Verfahren sind nicht linear, was soviel bedeutet, dass sich der Aufwand nicht direkt aus der einfachen Problemgröße ableiten lässt.

Weiterhin kann man ablesen, dass Bubblesort fast durchgehend _____ ist als Selection-Sort: Der Aufwand pro Karte ist bei Bubblesort ungefähr _____wie bei Selection-Sort.

Insgesamt ist der Aufwand pro Karte bei beiden Verfahren linear zur Problemgröße. Es gilt etwa:

Verfahren	**Abhängigkeit des Aufwands pro Karte von der Problemgröße**
Selection-Sort	≈ • *Problemgröße*
Bubble-Sort	≈ • *Problemgröße*

Da sich der **Gesamtaufwand** berechnen lässt durch *Gesamtaufwand = Aufwand pro Karte • Anzahl Karten,* lässt sich also leicht einsehen, dass der Gesamtaufwand beider Verfahren vom Quadrat der Problemgröße abhängt:

Verfahren	**Abhängigkeit des Gesamtaufwandes von der Problemgröße**
Selection-Sort	≈ • *Problemgröße²*
Bubble-Sort	≈ • *Problemgröße²*

Beim Vergleich von Algorithmen lässt man normalerweise konstante Faktoren weg. Dadurch wären **beide Sortierverfahren gleichwertig, nämlich quadratisch.**

Bleibt nur noch die Frage, warum lässt man Konstanten weg?

Nicht nur in der Informatik, auch in anderen Wissenschaften sucht man immer möglichst einfache Beschreibungsmöglichkeiten. Eine kurze, prägnante Formel ist immer vorzuziehen gegenüber einem komplizierten und langen Ausdruck. Wichtig ist, dass alle wesentlichen Aspekte berücksichtigt wurden.

Was ist nun am Aufwand eines Algorithmus wesentlich?

Konstante Faktoren können durch die Wahl eines schnelleren Computers ausgeglichen werden. Sind zum Beispiel 1.000.000 Karten zu sortieren, bräuchte Seletion-Sort ungefähr _____ Milliarden Elementaroperationen, Bubblesort _____ Milliarden. Selbstverständlich würde man in der Praxis den schnelleren Algorithmus einsetzen. Andererseits kann das Manko jederzeit durch Einsatz eines schnelleren Rechners ausgeglichen werden. Für bestimmte Gegebenheiten mag sogar Bubblesort besser geeignet sein: Es werden immer nur direkt benachbarte Speicherstelle miteinander verglichen. Denkt man daran, dass die Daten auf einer Festplatte liegen, geht also der Vergleich ohne große Bewegung des Schreib-/Lesekopfes vonstatten.

Wesentlich für einen Algorithmus ist es aber, wie stark der Zeitaufwand anwächst, wenn die Problemgröße wächst! Dieses Verhältnis lässt sich nicht durch Wahl eines stärkeren Rechners verändern - übrigens genauso wenig wie durch die Feinjustage (kleine Verbesserung) des Algorithmus.

Wenn wir zukünftig also über die Laufzeitkomplexität von Algorithmen unterhalten werden, interessiert uns immer nur das grobe Verhältnis zwischen Laufzeit und Problemgröße:

Die Aufwandsabschätzung / Komplexitätsanalyse:

Mit der Aufwandabschätzung bezüglich der Laufzeit eines Algorithmus (Laufzeitkomplexität) wird in der Informatik oft die Qualität von Algorithmen bestimmt. Diese Abschätzung gibt an, wie stark die notwendigen Rechenzeit in Bezug zur Problemgröße anwächst. Konstante Faktoren werden dabei nicht berücksichtigt.

Konsequenzen:

- Durch Weglassen der konstanten Faktoren kann die tatsächliche Laufzeit eines „schlechten" Algorithmus bei kleinen Problemgrößen durchaus kleiner sein als bei einem „guten" Algorithmus.
- Die Laufzeit ist unabhängig vom verwendeten Rechnermodell, der eingesetzten Programmiersprache und deren Implementierung oder der Taktfrequenz der CPU.
- Es bleibt ohne Bedeutung, wie viel Zeit ein Rechner für eine so genannte "Basis-Operation" (Wertzuweisung/if-Abfrage/etc.) in Anspruch nimmt, ob der Algorithmus also auf einem Supercomputer oder einem PC läuft - es soll nur untersucht werden, wie sich der Algorithmus bei n Eingabewerten verhält, wie viele Basis-Operationen in diesem Fall ausgeführt werden müssen.
- Uns interessiert eigentlich vor allem eine oberen Schranke für den Algorithmus: d.h. dass das angegeben Verhältnis (hier beim Sortieren „quadratisch") stets ÜBER der eigentlichen Laufzeitfunktion liegen sollte, wenn die Anzahl der Eingabewerte einmal einen bestimmten Wert überschritten haben.

Angabe der Laufzeiten mit der Groß-O-Notation:

Zur Angabe von Laufzeitkomplexitäten hat sich die O-Notation (engl.: *Big-Oh-Notation*) durchgesetzt, welche ein elegantes Hilfsmittel zur mathematischen Beschreibung der Laufzeit eines Algorithmus' darstellt.

Für unsere beiden Sortier-Verfahren (Selection-Sort" und „Bubble-Sort") würde man sagen, dass sie die Laufzeit $O(n^2)$ (sprich: „O von n Quadrat") besitzen, wobei n die Problemgröße, hier also die Anzahl zu sortierender Karten, bezeichnet und man also zum Ausdruck bringt, dass die Laufzeit quadratisch von der Problemgröße abhängt.

Historisches
Eingeführt wurde die Symbolik der O-Notation durch den deutschen Mathematiker Paul Bachmann (1837-1920) im Jahr 1894 in seinem Werk "Analytische Zahlentheorie". Später machte der Zahlentheoretiker Paul Landau (1877-1938) davon Gebrauch, weshalb die O-Notation gern auch als Schreibweise *Landau'scher Symbole* bezeichnet wird...

Grundgedanken/Ziele
Grundgedanken und Ziele der O-Notation haben wir am Beispiel der Sotierverfahren „Selection-Sort" und „Bubble-Sort" schon erarbeitet.

Kurz zusammenfasst sind dies i.A. drei Grundgedanken (siehe hier auch unter „Konsequenzen" auf der vorherigen Seite):

- der erste ist die **Betrachtung der Laufzeit in Abhängigkeit der Eingabedaten** des Algorithmus'. Es soll also eine Funktion f in Abhängigkeit der Anzahl der Eingabewerte n gefunden werden, welche die Laufzeit eines Algorithmus' wiedergibt: f(n).
- Zum Zweiten sollen **"unwesentliche" Konstanten außer Acht** gelassen werden. Die angegebene Funktion f(n) soll unabhängig sein von systemspezifischen Details (tatsächlicher Zeitaufwand), daher ist von konstanten Faktoren c zu abstrahieren.
- Der dritte Grundgedanke richtet sich an die **Untersuchung einer oberen Schranke**. Es soll also eine Untersuchung des "ungünstigsten Falls" stattfinden und die Funktion f(n) soll mit einer Konstanten c als Faktor eine Asymptote bzw. Majorante zur tatsächlichen Laufzeit g(n) darstellen.

Aus dem dritten Grundgedanken heraus, könnte man auf die Idee kommen, zum Beispiel für „Bubble-Sort" die Funktion $f(n)=n^3$ als obere Schranke zu nehmen. Das macht keinen Sinn. **Man such also nicht *irgendeine* Funktion**, welche die tatsächliche Laufzeit majorisiert, **sondern die *kleinste***.

Allgemein sucht man also eine Funktion f(n), für welche gilt: "$c*f(n) \geq g(n)$", wobei g(n) die eigentliche Laufzeit und c die systemabhängige Konstante bezeichnet. Ist c*f(n) die kleinste Majorante von g(n), so kann dieser Term auch "Asymptote" von g(n) genannt werden, womit die Angabe der Laufzeit mittels O-Notation oft auch als "***asymptotische Laufzeitkomplexität***" bezeichnet wird.

Der O-Notation nicht blind vertrauen!
Auch wenn die O-Notation ein wichtiges mathematisches Hilfsmittel zur Angabe oberer Laufzeitschranken darstellt, so ist sie doch keine "eierlegende Wollmilchsau". Vielmehr muss man an einigen Stellen aufpassen, nicht zu sorglos damit umzugehen und allein auf diese Schreibweise zu vertrauen. Insgesamt sollte man zumindest drei Faktoren berücksichtigen:

a. Die O-Notation gibt eine "obere Schranke" an, das heißt aber nicht, dass der Algorithmus auch jemals so viel Zeit benötigt, es heißt nur, dass garantiert NIE mehr Zeit benötigt wird. Vielleicht tritt der untersuchte ungünstigste Fall in der Praxis nie auf. Deshalb sollte man immer auch noch eine Betrachtung des durchschnittlichen und besten Falls in Erwägung ziehen.
b. Die systemabhängige Konstante c ist im Allgemeinen unbekannt, muss aber nicht unbedingt klein sein. Dies birgt ein erhebliches Risiko. Z.B. würde man natürlich einen Algorithmus, welcher n^2 Nanosekunden benötigt, einem solchen vorziehen, der n Jahrhunderte läuft, anhand der O-Notation könnte man diese Entscheidung allerdings nicht treffen, da eine lineare Komplexität besser zu sein scheint als eine quadratische. Hier ist also höchste Vorsicht geboten!
c. Schließlich ist die O-Notation eine Abschätzung der Laufzeit bei einer unendlichen Eingabemenge. Da jedoch keine Eingabe unendlich ist, sollte man bei der Wahl von Algorithmen die realistische Eingabelänge mit einbeziehen.