Algorithmen & Komplexität

Theoretische Informatik

Roman Wetenkamp

6. Mai 2021

Inhaltsverzeichnis

I. Grundlagen der Algorithmik **4**

1. Der Algorithmusbegriff **4**

2. Effizienz / Komplexität **5**

3. Maß der Komplexität: O-Notation **6**
3.1. Ω-Notation 7
3.2. Θ-Notation 8
3.3. Zusammenfassung 8

4. Rekursionsgleichungen **8**
4.1. Mastertheorem 8
4.2. Auflösungsverfahren 9

5. Aufgaben **11**

II. Suchalgorithmen **12**

6. Lineare Suche **12**

7. Binäre Suche **13**

8. Textsuche **14**
8.1. Einfache Textsuche 14
8.2. Knuth-Morris-Pratt-Algorithmus (KMP) 15

III. Anhang **15**

9. Lösungen **15**
9.1. Grundlagen der Algorithmik 15

10. Tabellen und Formeln **15**

Vorwort

Die theoretische Informatik ist genau das Thema, vor dem ich vor Beginn meines Studiums am meisten Respekt hatte ... Und das ist es neben den anderen Mathematik-Vorlesungen bis heute: Theoreme, Kalküle und Formeln sind nun einmal nicht die Dinge, mit denen sich die Masse der Informatikstudierenden gerne befasst, mich eingeschlossen. Mit einer (vermutlich) gerade eben bestandenen Logikklausur im ersten Semester sind dies Gründe genug dafür ergänzendes, motivierendes und begreifbares Material zusammenzustellen, dass mich einerseits akut durch die Klausur dieses Semesters bringt und hoffentlich auch für andere einen Nutzen hat. Geteiltes Leid ist halbes Leid!

Hier möchte ich den Vorlesungsstoff auf meine Art zusammenfassen und komplettieren, um Beispiele ergänzen und mit Aufgaben versehen, wie ich sie in der Vorlesung oder weiteren Büchern antraf. Ich persönlich lerne durch Aufgaben einfach am besten und für diejenigen, denen es auch so geht, gibt es davon hier ausreichend. Lösungen finden sich im Anhang.

Viel Erfolg!

Roman Wetenkamp
Mannheim, den 6. Mai 2021

Teil I.
Grundlagen der Algorithmik

1. Der Algorithmusbegriff

Wie der Titel dieses Moduls schon verrät, befassen wir uns mit Algorithmen: Jenen nebulösen Dingen, die uns mit ähnlich Denkenden und Interessierten in Social-Media-Plattformen zusammenbringen, komplexe mathematische Berechnungen ausführen und zunehmend mehr an Einfluss in unserem Leben gewinnen. Wir betrachten hier nun die Wortherkunft, definieren den Begriff anschließend und gehen auf Eigenschaften eines solchen ein.

Wortherkunft Der Begriff „Algorithmus“ besteht zum einen aus dem altgriechischen Wort *arithmos* – Zahl und zum anderen geht er zurück auf den persischen Mathematiker AL-CHARISMI (780-846 n. Chr.), dessen Werk *„Algorismus“* schon eine gewisse Nähe zum heute üblichen Begriff offenbart. [Enz20]

Definition Unter einem Algorithmus verstehen wir eindeutige Verarbeitungsvorschriften zur Lösung eines Problems oder einer Problemklasse. Daran geknüpft ist der Gedanke des EVA-Prinzips: Eine Eingabe wird verarbeitet und auf Grundlage dessen wird eine Ausgabe erzeugt, ein Algorithmus ist hier für die Verarbeitung bis zur Ausgabe notwendig.
Algorithmen finden in einer Vielzahl von Bereiches unseres Lebens Einsatz finden:

- Zur Untersuchung großer Datenmengen
- Zur Kommunikation/Suche im Internet
- In bildgebenden Verfahren oder diagnostischen Anwendungen der Medizin
- Assistenzsystemen im Auto
- Bei der Partnersuche über Online-Dating-Plattformen

Damit finden wir intuitiv eine Begründung für die Untersuchung von Algorithmen.

Bemerkung 1. *Ein Algorithmus ist von einem Programm unbedingt abzugrenzen: Der Algorithmus bezeichnet ein abstraktes Konzept zur Lösung eines gegebenen Problems während ein Programm die konkrete Umsetzung eines Algorithmus in einer Programmiersprache ist.*

Bei der Betrachtung von Algorithmen betrachten wir die folgenden drei Ziele:

- Korrektheit
- Effizienz
- Einfachheit

Einfachheit Aus wirtschaftlichen Gründen besteht ein Interesse daran, dass ein Algorithmus so einfach wie möglich ist, beispielsweise in Bezug auf Implementierungskosten. Ein weiterer Aspekt besteht in der Prüfung auf Korrektheit: Zu einem zu komplexen Algorithmus lässt sich nur schwer feststellen, ob dieser korrekt ist.

Korrektheit Die Korrektheit ist die zentrale Eigenschaft eines Algorithmus – Wenn ein Algorithmus nicht das leistet, was er vorgibt zu leisten oder was von ihm erwartet wird, ist er wertlos. Die Spezifikation beschreibt das exakte Verhalten unter Angabe von einer Vorbedingung (Zustand der Daten vor der Ausführung) und einer Nachbedingung (erwünschter Zustand nach der Ausführung).

Definition 1. *Ein Algorithmus heißt **partiell korrekt**, falls aus erfüllter Vorbedingung die Nachbedingung folgt.*
*Ein Algorithmus heißt **total korrekt**, falls er partiell korrekt ist und nach endlich vielen Schritten terminiert.*

Die Korrektheit eines Algorithmus kann manuell, per Implementierung oder mathematisch verifiziert werden, wobei letztere Variante häufig vorzuziehen ist.

Bemerkung 2. *Nicht jeder Algorithmus muss für jede Eingabe terminieren. Unter dem **Halteproblem** versteht man die Frage, ob ein Algorithmus nach endlich vielen Schritten terminiert. Diese Frage ist algorithmisch nicht entscheidbar, wie* Alan Turing *feststellte. Stattdessen muss die Termination für jeden Algorithmus einzeln entschieden werden.*

2. Effizienz / Komplexität

Weitaus komplexer als die Frage der Einfachheit oder Korrektheit ist jene der Komplexität selbst:

Definition 2. *Die Effizienz / Komplexität eines Algorithmus ist ein Maß für die Menge an Ressourcen (Rechenzeit oder Speicherbedarf), die er benötigt.*

Dabei ist zu beachten, dass es für ein Problem häufig mehrere Algorithmen gibt, die sich in ihrer Komplexität unterscheiden. Daher muss die Komplexität für jeden Algorithmus einzeln bestimmt werden. Dieses Skript befasst sich im Wesentlichen nur mit der Rechenzeit.

Für eine wirklich präzise Aussage über die Rechenzeit wäre es erforderlich, für jede Implementation eines Algorithmus in einer Programmiersprache einzeln festzustellen, welche Rechenzeit jede einzelne Operation auf dem spezifischen Prozessor des Testsystems benötigt. Da diese Aussagen mittlerweile aufgrund der Entwicklungen auf dem Prozessormarkt an Relevanz verlieren, verlagert sich die Betrachtung auf die Komplexität eines Algorithmus.

Die Komplexität eines Algorithmus kann folglich keine genaue Angabe in einer Zeiteinheit sein, die beispielsweise aussagt, wie schnell denn nun wirklich der größte gemeinsame Teiler von 443 und 123 mithilfe des euklidischen Algorithmus berechnet werden kann, sondern ist ein abstraktes Konzept, das Vergleiche zwischen Algorithmen ermöglichen soll:

- Abstraktion konstanter Faktoren
- Abstraktion unbedeutender Terme
- Wachstumsverhalten der Laufzeit bei Veränderung der Größe der Eingabe

Diese Abstraktion motiviert nun den folgenden Abschnitt.

3. Maß der Komplexität: O-Notation

Nach dem Mathematiker Edmund Landau definieren wir in Bezug auf das Wachstum von Funktionen eine „obere Schranke“ und nennen diese $O(f)$. Falls eine Funktion g nicht schneller wächst als f, so sagen wir, dass $g \in O(f)$. Eine ebenfalls übliche, jedoch irritierende Schreibweise ist $g = O(f)$.

Definition 3. *Sei* $\mathbb{R}_+ := \{x \in \mathbb{R} \mid x > 0\}$ *und* $\mathbb{R}_+^{\mathbb{N}} := \{f \mid f \text{ist eine Funktion der Form } f : \mathbb{N} \to \mathbb{R}_+\}$

Sei $g \in \mathbb{R}_+^{\mathbb{N}}$ *gegeben. Dann ist die Menge aller Funktionen, die höchstens so schnell wachsen wie* g *definiert als:*

$$O(g) := \{f \in \mathbb{R}_+^{\mathbb{N}} \mid \exists n_0 \in \mathbb{N} : \exists c \in \mathbb{R}_+ : \forall n \in \mathbb{N} : (n \geq n_0 \Rightarrow f(n) \leq c \cdot g(n))\}$$

Die O-Notation gibt folglich eine Menge vergleichbarer Funktionen an, die es ermöglichen, die Komplexität eines Algorithmus abzuschätzen.

Beispiel 1. *Behauptung:* $3n^3 + 2n^2 + 7 \in O(n^3)$

Beweis. Gesucht ist eine Konstante $c \in \mathbb{R}_+$ und eine Konstante $n_0 \in \mathbb{N}$, sodass für alle $n \in \mathbb{N}$ mit $n \geq n_0$ gilt:
$3n^3 + 2n^2 + 7 \leq c \cdot n^3$

Wähle $n_0 := 1$ und $c := 12$ [1]
Sei nun

$$\begin{aligned}
& 1 \leq n && (1)\\
(1)^3 : \quad & 1 \leq n^3 && (2)\\
(2) \cdot 7 : \quad & 7 \leq 7n^3 && (3)\\
(1) \cdot 2n^2 : \quad & 2n^2 \leq 2n^3 && (4)\\
& 3n^3 \leq 3n^3 && (5)\\
(3) + (4) + (5) : \quad & 3n^3 + 2n^2 + 7 \leq 12n^3 && (6)
\end{aligned}$$

Durch die Ungleichungen (3), (4) und (5) ist für jedes Polynom einzeln eine Abschätzung erfolgt. Addiert man diese Ungleichungen nun, ergibt sich (6). Damit ist gezeigt, dass die Behauptung wahr ist und die Funktion in $O(n^3)$ liegt. Aus der Definition der O-Notation ergibt sich, dass jede Funktion mit höchstem Grad 3 in $O(n^3)$ liegt, da Vorfaktoren und Konstantglieder entfallen. □

Die O-Notation eines Algorithmus lässt sich auch durch vollständige Induktion beweisen.

Beispiel 2. *Behauptung:* $n \in O(2^n)$

Beweis. Wähle $n_0 := 0, c := 1$. Zu zeigen: $n \leq 2^n \forall n \in \mathbb{N}$

Beweis durch Induktion nach n:
Induktionsanfang (IA): $n = 0 \quad n = 0 \leq 1 = 2^0 = 2^n$
Induktionsvoraussetzung (IV): $n \leq 2^n$ z.Z. $n + 1 \leq 2^{n+1}$

[1] Die Wahl von c und n erfolgt hier durch Ausprobieren – In diesem Fall ergibt sich 12 als Summe aus den Potenzfaktoren.

Induktionsschritt (IS): $n \mapsto n+1$
Per einfacher Induktion kann man nun zeigen:

$$n \leq 2^n \tag{7}$$

$$\text{Daraus folgt mit IV:} \quad n+1 \leq 2^n + 2^n = 2 * 2^n = 2^{n+1} \tag{8}$$

□

Die Eigenschaft $f \in O(g)$ induktiv zu zeigen, ist mühsam. Stattdessen können die folgenden Propositionen genutzt werden:

Bemerkung 3.

$$f \in O(f) \tag{9}$$

$$f \in O(g) \Rightarrow d \cdot f \in O(g) \tag{10}$$

$$f \in O(n) \wedge g \in O(n) \Rightarrow f + g \in O(n) \tag{11}$$

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 \cdot f_2 \in O(g_1 \cdot g_2) \tag{12}$$

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow \frac{f_1}{f_2} \in O(\frac{g_1}{g_2}) \tag{13}$$

$$f \in O(g) \wedge g \in O(n) \Rightarrow f \in O(n) \tag{14}$$

Darüber hinaus gilt folgender Satz:

Satz 1. *Seien $f, g : \mathbb{N} \to \mathbb{R}_+$. Dann gilt:*

$$f(n) \in O(g(n)) \iff (\frac{f(n)}{g(n)})_{n \in \mathbb{N}} \textit{ ist beschränkt}$$

Außerdem finden Grenzwertbetrachtungen Eingang:

Lemma 1.

$$f \in O(g) \textit{ und } g \in O(f), \textit{ wenn } \lim_{n \to \infty} (\frac{f(n)}{g(n)}) = c, c \neq 0 \tag{15}$$

$$f \in O(g) \textit{ und } g \notin O(f), \textit{ wenn } \lim_{n \to \infty} (\frac{f(n)}{g(n)}) = 0 \tag{16}$$

$$f \notin O(g) \textit{ und } g \in O(f), \textit{ wenn } \lim_{n \to \infty} (\frac{f(n)}{g(n)}) = \infty \tag{17}$$

Neben der O-Notation gibt es zwei weitere Notationen, die im Folgenden vorgestellt werden.

3.1. Ω-Notation

Neben der Menge aller Funktionen, die höchstens so schnell wächst wie die betrachtete Funktion, bezeichnen wir die Menge der Funktionen, die mindestens so schnell wächst wie die Funktion als $\Omega(n)$.

Definition 4. *Sei $g \in \mathbb{R}_+$. Die Menge aller Funktionen, die mindestens so schnell wachsen wie g ist:*

$$\Omega(n) := \{f \in \mathbb{R}_+^{\mathbb{N}} \mid \exists n_0 \in \mathbb{N} : \exists c \in \mathbb{R}_+ : \forall n \in \mathbb{N} : (n \geq n_0 \Rightarrow F(n) \geq c \cdot g(n))\}$$

Diese Notation fristet ein Schattendasein: In der Regel ist die O-Notation geeigneter, da eine Orientierung am „Worst-Case-Szenario“ üblicher ist.

3.2. Θ-Notation

Aus O-Notation und Ω-Notation ergibt sich nun die Menge der Funktionen, die genau so schnell wachsen wie die betrachtete Funktion. Wir bezeichnen sie mit Θ.

Definition 5.

$$\Theta(g) = O(g) \cap \Omega(g)$$

3.3. Zusammenfassung

- Die O-Notation umfasst alle Funktionen, die nicht schneller wachsen als f. Damit beschreiben wir eine obere Schranke für die Komplexitätsfunktion.
- Die Ω-Notation umfasst alle Funktionen, die mindestens so schnell wachsen wie f. Dies ist eine untere Schranke.
- Die Θ-Notation umfasst alle Funktionen, die genau so schnell wachsen wie f. Damit haben wir ein asymptotisches Maß.

Lemma 2.

- $f \in O(g)$ *und* $g \in O(f)$, *also* $f \in O(g) und f \in \Omega(g)$, *wenn* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c, c \neq 0$, *also:* $f \in \Theta(g)$
- $f \in O(g)$ *und* $g \notin O(f)$, *wenn* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, *also:* $g \in \Omega(f)$ *und* $f \notin \Omega(g)$
- $f \notin O(g)$ *und* $g \in O(f)$, *also* $g \notin \Omega(f)$ *und* $f \in \Omega(g)$, *wenn* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

4. Rekursionsgleichungen

Nun haben wir mit den Landau-Symbolen einen mathematischen Weg gefunden, die Komplexität von Algorithmen zu beschreiben. Im Folgenden wenden wir dieses Modell auf konkrete Algorithmen an.

Bekannte Algorithmen sind beispielsweise Such- oder Sortierverfahren. Diese sind häufig rekursiv, d.h. eine Iteration oder verarbeitet direkt das Ergebnis einer vorherigen. Dadurch ergeben sich nun sogenannte **Rekurrenzgleichungen** oder auch **Rekursionsgleichungen**.

Beispiel 3. $T(n) = 2 \cdot T(n-) + 1$ *ist eine Rekurrenzgleichung, da der Funktionswert n direkt vom Funktionswert* $n-1$ *abhängt. Aus den Programmierkonzepten ist uns Rekursion als ein „Wiederaufruf von sich selbst“ bekannt, dieses Muster findet sich hier in unseren betrachteten Algorithmen.*

Um die Komplexität eines rekursiven Algorithmus bestimmen zu können, bedienen wir uns je nach Rekursionsgleichung unterschiedlichen Verfahren. Eines davon ist das sogenannte Mastertheorem.

4.1. Mastertheorem

Satz 2. *Sofern alle Teilprobleme die gleiche Größe haben, können wir zwei Fälle unterscheiden:*

- *Basisfall:* $f(n)$ *ist konstant für hinreichend kleine Werte von* n

- *Für größere n lässt sich eine Rekursionsgleichung der folgenden Form bestimmen:*

$$f(n) = a \cdot f(\frac{n}{b}) + O(n^d)$$

a ist hier die Anzahl der rekursiven Aufrufe, b der Verkleinerungsfaktor des Problems und d der Exponent für die Laufzeit der Vorbereitung der Rekursion und/oder der Zusammensetzung der Teilprobleme beschreibt.

Nun gilt Folgendes:

$$f(n) \in \begin{cases} O(n^d) & \text{falls } a < b^d \\ O(n^d \cdot log_b(n)) & \text{falls } a = b^d \\ O(n^{log_b(a)}) & \text{falls } a > b^d \end{cases}$$

Mithilfe dieses Theorems kann nun die Komplexität eines rekursiven Algorithmus bestimmt werden:

Beispiel 4. *Gegegeben sei*

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

Daraus folgt $a = 2, b = 2, d = 2$.

Da $2 < 2^2$ *trifft der erste Fall ein und wir halten fest:* $f(n) \in O(n^2)$.

Über das Mastertheorem sind eine Zahl an Rekursionsgleichungen entscheidbar, jedoch nicht jede. Für die Rekursionsgleichungen, deren Form nicht der für das Mastertheorem benötigten entspricht, werden andere Verfahren nötig.

4.2. Auflösungsverfahren

Eine gegebene Rekursionsgleichung lässt sich in eine geschlossene Form überführen, in der keine Abhängigkeit zu vorherigen Funktionswerten mehr besteht. Dafür existieren zwei Ansätze:

- **bottom-up:** Aus berechneten Funktionswerten für kleine n wird eine weitere Berechnungsvorschrift ermittelt. Diese muss anschließend induktiv bewiesen werden.
- **top-down:** Die vorherigen Funktionswerte werden solange durch die entsprechende Gleichung ersetzt, bis der definierte Startwert eingesetzt werden kann. Anschließend wird der Term zusammengefasst und es ergibt sich eine geschlossene Rechenvorschrift.

Beispiel 5. *Gegeben ist die folgende Rekursionsgleichung:* $f(1) = 1, f(n) = 2 \cdot f(n-1) + 1$.

bottom-up

$$f(1) = 1 \tag{18}$$
$$f(2) = 2 \cdot 1 + 1 = 3 \tag{19}$$
$$f(3) = 2 \cdot 3 + 1 = 7 \tag{20}$$
$$f(4) = 2 \cdot 7 + 1 = 15 \tag{21}$$

Aus den Ergebnissen wird relativ schnell eine Verwandtschaft zu den ersten Potenzen der Zahl 2 deutlich und so lässt sich nun folgende Gleichung aufstellen:

$$f(n) = 2^n - 1$$

Diese Gleichung kann nun für die gegebenen Werte von n überprüft werden und muss anschließend induktiv bewiesen werden.

top-down

$$\begin{aligned}
f(n) = &\quad 2 \cdot f(n-1) + 1 && (22)\\
= &\quad 2 \cdot (2 \cdot f(n-2) + 1) + 1 && (23)\\
= &\quad 2 \cdot (2 \cdot (... \cdot (2 \cdot 1) + 1)...) + 1) + 1 && (24)\\
\Rightarrow &\quad 2^{n-1} + 2^{n-2} + ... + 2^{n-(n-1)} + 1 && (25)\\
= &\quad \sum_{i=0}^{n-1} 2^i = \frac{2^{(n-1)+1} - 1}{2 - 1} = 2^n - 1 \in O(2^n) && (26)
\end{aligned}$$

Je nach Rekursionsgleichung bietet sich mitunter eines der beiden Verfahren mehr an als das andere. Vernachlässigt werden darf beim *bottum-up*-Verfahren nicht, dass ein Induktionsbeweis erforderlich ist! Bei Rekursionsgleichungen mit zweifacher Rekursion (in Beziehung zu den vorherigen zwei Funktionswerten) ist ein *top-down*-Verfahren deutlich komplexer und häufig nicht ratsam.

5. Aufgaben

Aufgabe 1 Veranschaulichen Sie die O-, Θ- und Ω-Notation anhand der Funktion $f(n) = 3n^3 + 7n^2 + 16$ grafisch.

Lösung auf Seite 15

Aufgabe 2 Geben Sie zu folgenden Funktionen jeweils die O-Notation an.

1. $f(n) = 6n^4 + 3 \cdot log_2(n)$
2. $f(n) = 6627816n + 13$
3. $f(n) = \frac{72n^3+27n^2+8n+9}{n!}$
4. $f(n) = 3 \cdot f(n-1) + \frac{f(n-2)}{n^2}$

Lösung auf Seite 15

Aufgabe 3

1. Zeigen Sie, dass $n^2 \in O(2n)$.
2. Zeigen Sie, dass $n^3 \in O(2n)$.

Lösung auf Seite 15
entnommen aus VL

Aufgabe 4 Bestimmen Sie unter Anwendung des Mastertheorems die jeweilige Komplexitätsklasse $O(n)$.

- $f(n) = 2 \cdot f(\frac{n}{2n+1}) + n^2$
- $g(n) = log(n) \cdot g(\frac{n}{2}) + 3$
- $h(n) = sin(n) \cdot h(\frac{n}{\frac{n}{2}}) + O(n^3)$

Lösung auf Seite 15

Versionshinweis Weitere Aufgaben zu Rekursionsgleichungen, praktischen Anwendungen auf Pseudo-Code und für Θ, Ω folgen in späteren Versionen

Teil II.
Suchalgorithmen

Sie studieren Informatik, sind es satt, noch mehr Stunden vor Bildschirmen zu verbringen und suchen sich deshalb ein echtes, physisches Buch aus der Bibliothek Ihrer Hochschule heraus. Sie fragten nach „Wirth: Algorithmen und Datenstrukturen“ von 1976, weil man es Ihnen in Ihrer Vorlesung empfahl, die Bibliothekarin deutet auf ein Regal und lässt sie damit allein.

Wie finden Sie das gesuchte Buch aus der Regalreihe mit etwa 300 Büchern?

6. Lineare Suche

Sie beginnen ganz links. Nun lesen Sie den Titel des ersten Buches, er passt nicht. Sie lesen den Titel des zweiten Buches, er passt ebenfalls nicht. Sie fahren fort, iterieren über das gesamte Bücherregal und brechen ab, falls Sie es gefunden haben oder am Ende angelangt sind. Dieses Verfahren ist die **Lineare Suche**.

Idee Durchlaufe sukzessive das Feld A, bis das gesuchte Element gefunden ist bzw. das Ende des Feldes erreicht ist.

```
LinearSearch(A[], E) {
        i = 0;
        While i < A[].length {
                If (A[i] == E) {
                        return i;
                }
                i = i + 1;
        }
        Return ElementNotFound
}
```

Komplexität Ganz offensichtlich ist der Aufwand dieses Suchverfahrens durch die Feldlänge (also die Anzahl der Bücher in unserem Regal) bestimmt.
günstigster Fall Das gewünschte Element steht an Indexposition 0 (also dem ersten Element bzw. es ist das erste Buch im Regal) → ein Vergleich ist erforderlich
durchschnittlicher Fall Das gewünschte Element steht an der mittleren Indexposition $\frac{n}{2}$ bei n Feldelementen. → $\frac{n}{2}$ Vergleiche sind erforderlich
ungünstigster Fall Das gewünschte Element ist nicht im Feld vorhanden. → n Vergleiche

Insgesamt ergibt sich – da wir bei der Angabe der Komplexität durch die O-Notation jeweils den ungünstigsten Fall betrachten – für die Lineare Suche eine Komplexität von $O(n)$.

7. Binäre Suche

Sie befinden sich in der perfekten Bibliothek: Alle Bücher sind aufsteigend nach ihren Autor*innen sortiert! Dieses Wissen können Sie sich zunutze machen, in dem Sie ihr Wunschbuch mit der **binären Suche** finden: Sie teilen die Bücherreihe in zwei Hälften und betrachten nun das mittlere Buch: Muss Ihr Buch links oder rechts von dem mittleren Buch stehen (Vergleich anhand des Namens der Autoren)? Oder ist es sogar ihr Buch? Nein, das ist es nicht und der Autor Ihres Buches hat einen im Alphabet weiter hinten stehenden Anfangsbuchstaben, also betrachten Sie die Hälfte rechts des mittleren Buches. Diese Hälfte teilen Sie nun erneut, betrachten die neue Mitte und fahren fort, bis ihre Hälften aus einem Buch bestehen.

Idee Vergleiche zu suchendes Element E mit der Mitte des Suchbereiches $A[m]$. Falls gefunden: Abbruch. Falls $E < A[m]$: Suche weiter in linker Hälfte. Falls $E > A[m]$: Suche weiter in rechter Hälfte. Das Weitersuchen erfolgt durch einen rekursiven Aufruf des Algorithmus mit der entsprechenden Hälfte des Feldes.

```
BinarySearch(A[], E, indexLeft, indexRight) {
        if (indexLeft > indexRight) {
                return ElementNotFound
        }
        else {
                mid = (indexLeft+indexRight) / 2;
                if (E == A[mid]) {
                        return mid;
                }
                else if (E < A[mid] {
                        return BinarySearch(A[], E, indexLeft, mid-1);
                }
                else {
                        return BinarySearch(A[], E, mid+1, indexRight);
                }
        }
}
```

Die Parameter `indexLeft` und `indexRight` schränken jeweils das zu betrachtende Feld ein. Zu Beginn ist `indexLeft` mit 0 und `indexRight` mit `A[].length()` initialisiert.

Komplexität Hier handelt es sich um ein rekursives Verfahren, folglich werden wir eine Rekursionsgleichung erhalten und diese mit den bekannten Verfahren auswerten. Doch zunächst betrachten wir wieder Fälle:
günstigster Fall: E ist das mittlere Element $\rightarrow$ 1 Vergleich
ungünstigster Fall: E ist nicht im Feld
Die Rekursionsgleichung mit den Parametern a, bund$d$ ergibt sich nun wie folgt:

- Es gibt einen rekursiven Aufruf, d.h. $a = 1$.
- Pro Rekursionsaufruf halbieren wir das Feld, daraus folgt $b = 2$.
- Die Vorbereitung der Rekursion verläuft konstant und ist minimal, also $d = 1$.

Die Rekursionsgleichung lautet nun:

$$f(n) = 1 \cdot f(\frac{n}{2}) + 1$$

Für die Anwendung des Mastertheorems formen wir um:

$$\iff f(n) = 1 \cdot f(\frac{n}{2}) + O(n^0)$$

Nun ergibt sich:

$$a = b^d \iff 1 = 2^0 \Rightarrow f(n) \in O(n^0 \cdot log_2(n)) = O(log(n))$$

8. Textsuche

Spätestens jetzt sollten Sie das Buch gefunden haben, wenn Sie die binäre Suche verwendet haben, vermutlich schneller als mit der linearen Suche. Jetzt haben Sie einen Arbeitsplatz gefunden, setzen sich und schlagen das Buch auf: Für ein Lesen in Ruhe haben Sie keine Zeit, daher suchen Sie sich die interessanten Teile heraus. Sie suchen, wo Niklaus Wirth Ihnen die O-Notation erklären wird.

Formal suchen Sie folglich alle Vorkommen eines Musters (in Form einer Zeichenkette) in einem Text. Dafür betrachten wir zwei Verfahren.

8.1. Einfache Textsuche

Analog zur linearen Suche beginnen Sie einfach, Ihren gesuchten Begriff unter den Text zu legen. Stimmt der erste Buchstabe überein, vergleichen Sie den zweiten. Stellen Sie eine Differenz fest, verschieben Sie Ihr Suchwort um eine Position nach rechts. Derart fahren Sie fort, bis Sie am Ende des Textes angelangt sind.

Idee Beginnend beim ersten Zeichen des Textes legt man das Muster der Reihe nach an jede Stelle des Textes an und vergleicht zeichenweise von links nach rechts, ob eine Übereinstimmung vorliegt

```
int BruteForceSearch(char[] text, char[] pattern) {
        int n = text.length;
        int m = pattern.length;
        int i = 0;
        int j = 0;
        for (int k = 0; i <= n-m; i++) {
                j = 0;
                while (j < m && Text[i+j] == pattern[j]) {
                        j++;
                }
                if (j == m) {
                        return i;
                }
        }
        return ElementNotFound;
}
```

Komplexität Offenbar hängt der Aufwand dieses Verfahrens ganz wesentlich davon ab, wie lang das Muster und der Text in Gänze sind. Weniger relevant hingegen ist der Aufbau des Textes, da alle Vorkommen des Musters gesucht werden. Insofern ist es nicht entscheidend, an welcher Stelle das Muster das erste Mal auftritt.

günstigster Fall Im günstigsten Fall werden m Vergleiche benötigt, wenn nämlich die Länge des Textes kleiner als das doppelte der Länge des Musters m ist und das Muster direkt am Beginn des Textes steht. $\rightarrow m$ Vergleiche

ungünstigster Fall Im ungünstigsten Fall werden $(n - m + 1) \cdot m$ Vergleiche benötigt, wobei dieser Fall allgemeingültig ist – es werden alle Vorkommen des Musters im Text gesucht und nicht lediglich das erste.

Die Laufzeit des Algorithmus liegt nun in $O(n \cdot m)$.

8.2. Knuth-Morris-Pratt-Algorithmus (KMP)

Teil III.
Anhang

9. Lösungen

9.1. Grundlagen der Algorithmik

10. Tabellen und Formeln