Effizienz von Such- und Sortieralgorithmen

Vorwissenschaftliche Arbeit verfasst von

Laurenz Weixlbaumer

Klasse 8iz

Betreuerin: Mag. Andrea Stiglbauer

BORG Linz

Honauerstraße 24

4020 Linz

Linz, 28. Februar 2020

Abstract

The analysis of algorithms has been perhaps one of the most influental fields of research in computer science over the last decades. Many fascinating unsolved problems remain in the area, as well as many solved ones. This paper aims to give an introduction to the analysis of algorithms without requiring extensive preliminary knowledge by first establishing some basic concepts and terminology of the field. A basic definition of both the word algorithm itself and a way to specify algorithms is followed by an introduction to the complexity of algorithms and its importance as well as the asymptotic analyis of algorithms and its utility. Advanced material such as a model of computation, the random access machine, and an in-depth definition of the Bachmann–Landau notation is included in later sections but not required to understand the main body of text. The paper concludes with an in-depth treatment of two real-world algorithms, the insertion sort, a basic but very illumnating sequential algorithm, and the binary search algorithm which makes use of the concept of binary trees to obtain a significant speedup.

Inhaltsverzeichnis

Einleitung

Diese Arbeit beschäftigt sich mit der Analyse der Effizienz von Algorithmen am Beispiel von Sortier- und Suchalgorithmen. Es werden Werkzeuge für eine solche Analyse eruiert und im Anschluss demonstriert.

Kapitel 1 führt die Begrifflichkeit „Algorithmus“ ein und legt dahingehend eine Methode zur Algorithmusspezifikation fest, auf die eine Definition des Begriffs im Kontext der Informatik folgt. Kapitel 2 bietet einerseits einen Einblick in die Motivation hinter der Analyse von Algorithmen und andereseits einen Überblick über verschiedene Begrifflichkeiten, Modelle und Methodiken in derselben.

Kapitel 3 und 4 beschäftigen sich jeweils mit dem Suchen und Sortieren, und gehen von einer grundsätzlichen Definition des jeweiligen Problems zur Beschreibung und anschließenden detaillierten Analyse eines repräsentativen Algorithmus über. Die Methodik hinter der Analyse von Algorithmen ist oftmals von Algorithmus zu Algorithmus sehr ähnlich; aus diesem Grund beschränken sich diese Kapitel auf die Analyse jeweils *eines* Algorithmus, der dann mit einem angebrachten Detailgrad behandelt werden kann.

Im Besonderen bildet die theoretische Analyse den Schwerpunkt der Bearbeitung des dargelegten Themas. In mit einem Stern (*) markierten Abschnitten und Unterabschnitten werden einige Themen näher beleuchtet als es vom Rest der Arbeit gefordert sein mag. Diese Abschnitte können übersprungen werden; sie sind für das Verständnis der folgenden Kapitel nicht von wesentlicher Bedeutung, durchaus jedoch

für das Feld der Analyse von Algorithmen!

1 Algorithmen

Noch um 1957 war „*algorithm*“ nicht in *Webster's New World Dictionary* vertreten. Dort war nur der ältere Begriff „*algorism*“ zu finden, zu Deutsch das Rechnen mit der arabischen Zahlschrift. *Algorism* entstammt dem Namen eines persischen Autors, *Abū 'Abd Allāh Muhammad ibn Mūsā al-Khwārizmī* [1](circa 825 A. D., vgl. Knuth, 1997, S. 1–2). Al-Khwārizmī schrieb das Buch *Kitab al jabr wa'l-muqābala* („[...] Rechenverfahren durch Ergänzen und Ausgleichen“), aus dessem Titel ein anderes Wort, „Algebra“, entstammt (Gericke, 1984, S. 197–199). Die Wandlung des Begriffs *algorism* zu *algorithm* (und damit Algorithmus) ist in *Vollständiges mathematisches Lexicon* (Wolff, 1747, S. 38) dokumentiert, hier wird der Begriff wie folgt definiert: „Unter dieser Benennung werden zusammen begriffen die 4 Rechnungs-Arten in der Rechen-Kunst nemlich addiren, multipliciren, subtrahiren und dividiren. [...]“ (alte Rechtschreibung übernommen).

Vor 1950 wurde der Begriff Algorithmus am häufigsten mit dem euklidischen Algorithmus[2]assoziiert, einem Verfahren zur Ermittlung des größten gemeinsamen Teilers zweier Zahlen (vgl. Knuth, 1997, S. 2).

Algorithmus E (*Euklids Algorithmus*). Gegeben seien zwei positive ganze Zahlen m und n. Der *größte gemeinsame Teiler*, also die größte positive ganze Zahl welche

[1]Al-Khwārizmīs Name stellt in der Literatur eine Quelle der Verwirrung dar: Horowitz, Sahni und Rajasekaran, 1997, S. 1, Knuth, 1997, S. 1, Pickover, 2009, S. 84 und Gericke, 1984, S. 197–199 verwenden bei ihren Erwähnungen jeweils unterschiedliche Namen, meinen jedoch die selbe Person.

[2]Der euklidische Algorithmus ist in Euklids *Elemente* (Buch 7, Satz 2) zu finden.

sowohl m als auch n gerade teilt, ist zu ermitteln.

E1 [Ermittle den Rest.] Dividiere m durch n, r seie der Rest. (Es gilt nun also $0 \leq r < n$.)

E2 [Ist er Null?] Gilt $r = 0$ so terminiert der Algorithmus, n ist die Lösung.

E3 [Verringern.] Setze $m \leftarrow n$, $n \leftarrow r$ und gehe zurück zu Schritt E1.

Zu Euklids Zeit, im Hellinismus, war eine Zahl bedeutungsgleich zu einer messbaren Länge – seine Definitionen, Postulate und Axiome waren überwiegend geometrische Natur. Tatsächlich wäre ihm die obige Darstellung gar nicht möglich gewesen, da Null (wie es in Schritt E2 verwendet wird) keine messbare Länge ist, und für ihn damit nicht verwendbar war (vgl. Callahan und Casey, 2015, S. 6).

Euklid selbst präsentierte seinen Algorithmus also nicht auf diese Weise. Dennoch illustriert das obig verwendete Format die Art und Weise, mit welcher alle im Laufe dieser Arbeit behandelten Algorithmen präsentiert und spezifiziert werden. Gegebenfalls wird begleitend ein *Programm*[3] in Pseudocode oder ein Ablaufdiagramm gegeben.

1.1 Spezifikation

Jeder Schritt eines Algorithmus, wie der obige Schritt E1, beginnt mit einem Ausdruck in eckigen Klammern, welcher den wesentlichen Inhalt des Schritts zusammenfasst. Darauf folgt eine Beschreibung (in Worten und Symbolen) einer auszuführenden *Anweisung* oder einer zu treffenden *Entscheidung*. Eingeklammerte *Kommentare* – wie der zweite Satz in Schritt E1 – tragen erklärende Informationen bei, sie geben keine Anweisungen.

Ein Pfeil $\leftarrow$, wie er in Schritt E3 zu sehen ist, stellt die *Zuweisung* dar: $m \leftarrow n$ bedeutet, dass der Wert der Variable m durch den Wert der Variable n zu ersetzen ist.

[3] Hier wird „Programm“ nach der Definition von Horowitz, Sahni und Rajasekaran, 1997, S. 2 verwendet: „A *program* is the expression of an algorithm in a programming language.“

Ein Pfeil wird verwendet, um Zuweisungsoperationen von Gleichheitsoperationen zu unterscheiden: „Setze $m = n$“ ist keine gültige Anweisung und wird nie vorkommen, „Ist $m = n$?“ jedoch möglicherweise schon. Das Zeichen $=$ beschreibt also eine Bedingung welche zu überprüfen ist, das Zeichen $\leftarrow$ eine Anweisung die ausführbar ist.

Das Verfahren, eine Variable n um eins zu erhöhen wird durch den Ausdruck $n \leftarrow n+1$ (zu lesen als „n wird ersetzt durch $n+1$“) angegeben. Im Allgemeinen wird bei Ausdrücken der Form „Variable $\leftarrow$ Formel“ erst die Formel mit den gegenwärtigen Werten der darin vorkommenden Variablen berechnet. Darauffolgend wird der Wert der Variable auf der linken Seite des Pfeiles mit dem eben errechneten Ergebnis ersetzt.

Die Reihenfolge der Anweisungen in Schritt E3 ist wichtig: „Setze $m \leftarrow n$, $n \leftarrow r$“ unterscheidet sich wesentlich zu „Setze $n \leftarrow r$, $m \leftarrow n$“ nachdem bei letzterer Anweisung der bisherige Wert der Variable n „verlorengeht“ bevor es für $m \leftarrow n$ verwendet werden kann. Folglich wäre letztere Anweisung äquivalent zu $n \leftarrow r$, $m \leftarrow r$. Ist mehreren Variablen der selbe Wert zuzuweisen werden mehrere Pfeile verwendet, $n \leftarrow r$, $m \leftarrow r$ wird als $n \leftarrow m \leftarrow r$ angeschrieben. Um die Werte zweier Variablen zu vertauschen, wird statt $t \leftarrow m, m \leftarrow n, n \leftarrow t$, wobei t eine neue Hilfsvariable ist, $n \leftrightarrow m$ geschrieben.

Ein Algorithmus beginnt bei dem am niedrigsten nummerierten Schritt (üblicherweise Schritt 1) und führt darauffolgende Schritte in aufeinanderfolgender Reihenfolge aus, sofern nicht anderweitig angegeben. In Schritt E3 macht die Anweisung „gehe zurück zu Schritt E1“ die Reihenfolge offensichtlich. Die Anweisung in Schritt E2 ist durch „Gilt $r = 0$“ eingeleitet. Gilt also $r \neq 0$ ist die restliche Anweisung zu ignorieren, hier hätte die redundante Anweisung „Gilt $r \neq$ gehe weiter zu Schritt E3“ hinzugefügt werden können.

Die fette vertikale Linie „▮“, am Ende von Schritt E3 zu sehen, kennzeichnet das

Ende des Algorithmus und die Forsetzung regulären Textes.

Die hier dargestellte Methode zur Algorithmusspezifikation ist angelehnt an die in Knuth, 1997, S. 3-5 verwendete Beschreibung von Algorithmen.

1.2 Definition

Die moderne Bedeutung des Wortes *Algorithmus* ist durchaus ähnlich zu *Rezept*, *Vorgang*, *Verfahren*, et cetera. Dennoch konnotiert das Wort etwas anderes – es ist nicht nur eine endliche Menge von Regeln, die eine Folge von Operationen für die Lösung einer bestimmten Aufgabe darstellen, ein Algorithmus hat nach Knuth, 1997, S. 4-6 zwingend folgende fünf Merkmale:

1. *Eingabe.* Ein Algorithmus hat keine, eine oder mehrere Eingangsmengen, welche entweder zu Beginn oder während der Laufzeit [?] gegeben werden.
2. *Ausgabe.* Ein Algorithmus hat eine oder mehrere Ausgabemengen, welche eine wohldefinierte Beziehung zu den Eingangsmengen haben.
3. *Eindeutigkeit.* Jeder Arbeitsschritt eines Algorithmus ist eindeutig definiert.
4. *Endlichkeit.* Ein Algorithmus terminiert[4] nach einer endlichen Anzahl von Arbeitsschritten.
5. *Effektivität.* Die Arbeitsschritte müssen einfach genug sein um in endlicher Zeit von einer Person mit Stift und Papier ausgeführt werden zu können.

Kriterien 1 und 2 verlangen, dass ein Algorithmus ein oder mehrere *Ausgaben* produziert und in der Lage ist, keine, eine oder mehrere von außen gelieferte *Eingaben* aufzunehmen. Laut Kriterium 3 muss jeder Arbeitschritt *eindeutig* sein, es muss also genau eine Möglichkeit geben die Anweisung auszuführen. Anweisungen wie „füge 3 oder 5 zu x hinzu“ oder „berechne $1/0$“ sind nicht erlaubt, nachdem entweder nicht

[4] Abbruch mit Erfolg oder Misserfolg

klar ist welche der zwei Möglichkeiten zu wählen ist ($3 + x$ oder $5 + x$) oder was die Lösung ist ($1/0$ ist nicht definiert).

Das vierte Kriterium setzt voraus, dass Algorithmen nach einer endlichen Anzahl von Schritten *terminieren*. Eine daran angelehnte Überlegung ist, dass das Kriterium der Endlichkeit für praktische Zwecke nicht stark genug ist. Ein nützlicher Algorithmus sollte nicht nur eine endliche Anzahl von Schritten haben, sondern eine *sehr* endliche Anzahl: eine vernünftige Anzahl (vgl. Horowitz, Sahni und Rajasekaran, 1997, S. 2)[5].

Kriterium 5 setzt für die Arbeitschritte nun nicht mehr nur die Eindeutigkeit (siehe Kriterium 3), sondern auch die *Realisierbarkeit* (im Sinne der Englischen *feasibility*) voraus. Eine hilfreiche Veranschaulichung ist hier die Voraussetzung, dass ein Arbeitsschritt in endlicher Zeit mit Stift und Papier lösbar sein muss. Operationen wie „$4 + 5$“ oder „ist $x \geq 0$“ sind realisierbar, nachdem ganze Zahlen endlich auf Papier gebracht werden können, und eine Methode zur Addition von ganzen Zahlen besteht (ein „Additionsalgorithmus“). Die selben Schritte wären jedoch nicht realisierbar, wenn die verwendeten Werte irrationale reelle Zahlen (beispielsweise π) wären. Ein weiteres Beispiel ist „Wenn 5 die größte ganze Zahl n ist für die gilt, dass $\alpha^n + \beta^n + \gamma^n = \delta^n$ sofern $\alpha, \beta, \gamma, \delta \in \mathbb{Z}$ dann gehe zu Schritt 2“. Eine solche Anweisung wäre nicht realisierbar bis ein Algorithmus, welcher feststellt ob die obige Bedingung wahr ist, konstruiert werden kann (vgl. Knuth, 1997, S. 6).

[5] Für ein Beispiel eines Algorithmus welcher eine interessantes Problemstellung in endlicher aber unrealistisch langer Zeit löst, siehe Knuth, 1997, S. 272–273, 28

2 Analyse

Sind für ein bestimmtes Problem mehrere Algorithmen verfügbar, so ist oft der „Beste“ zu ermitteln. Dieses Kapitel leitet grundlegende Bezeichnungen und Verfahren im Bereich der Analyse von Algorithmen ein.

In der Praxis sind nicht nur Algorithmen schlechthin gefragt, sondern Algorithmen die auf eine oft lose definierte ästhetische Art und Weise *gut* sind. Ein Kriterium der Güte eines Algorithmus ist die Dauer, die für die Ausführung eines Algorithmus benötigt wird (vgl. Knuth, 1997, S. 7). Andere Kriterien sind beispielsweise die Verständlichkeit beziehungsweise Einfachheit, die Qualität der begleitenden Dokumentation, oder die Funktionsfähigkeit des Algorithmus im Allgemeinen (vgl. Horowitz, Sahni und Rajasekaran, 1997, S. 13–15).

2.1 Einleitung

Zwei Algorithmen seien in Bezug auf ihre Effizienz zu vergleichen. Ein Weg dies zu tun ist beide Algorithmen als Programme zu implementieren und sie mit einer geeigneten Auswahl von Eingaben laufen zu lassen, während ihr Ressourcenverbrauch gemessen wird. Diese Herangehensweise ist aus vier Gründen oft unbefriedigend (vgl. Shaffer, 1997, S. 58):

1. Es wird Arbeitsaufwand für das Programmieren und Prüfen zweier Algorithmen aufgewendet, obwohl nur einer (im ungünstigsten Fall keiner) der beiden

schlussendlich verwendet wird.

2. Es ist außerordentlich schwierig beide Algorithmen „gleich gut“ zu programmieren. Ist einer der Algorithmen „besser“ programmiert als der andere, so repräsentieren die Programme nicht die relativen Qualitäten der Algorithmen die sie implementieren.
3. Die Wahl der Probleminstanzen[1] könnte einen der Algorithmen unfairerweise bevorteilen. Für eine genauere Behandlung des dargestellten Problems siehe Johnson, 1999, insbesondere S. 10–12.
4. Bei der Auswertung könnte klar werden, dass keiner der zwei Algorithmen dem Ressourcenbudget gerecht wird. In diesem Fall beginnt der Prozess mit neuen, andere Algorithmen implementierenden Programmen nochmals.

Diese Probleme können oft durch die Analyse von Algorithmen vermieden werden (vgl. Shaffer, 1997, S. 58). Einen Algorithmus zu analysieren bedeutet, die Ressourcen, welche der Algorithmus für seine Ausführung benötigt, zu prognostizieren (vgl. Cormen u. a., 2001, S. 21). Eine Spezialform dieses Vorgangs, die asymptotische Analyse von Algorithmen, prognostiziert den Ressourcenverbrauch eines Algorithmus während die Größe seiner Eingabemenge wächst. Sie liefert jedoch, im Gegensatz zur regulären Analyse von Algorithmen, keine Informationen über den relativen Wert zweier Algorithmen, die sich nur geringfügig in ihrer Effizienz unterscheiden (vgl. Bruijn, 1958, S. 1).

Die für ein Programm ausschlaggebende Ressource ist meistens seine Laufzeit, beziehungsweise seine „rechnende Zeit“ (vgl. Knuth, 1997, S. 97, Shaffer, 1997, S. 58). Eine vollständige Analyse kann jedoch auch andere Faktoren wie den Raum[2] oder Latenzzeit (vgl. Melani u. a., 2017) berücksichtigen.

Die Laufzeit eines Programms wird durch viele Faktoren beeinflusst. So hat zum

[1] „Probleminstanz“ oder nur „Instanz“ im Sinne von *problem instance*, kann grob als äquivalent zur Eingabemenge betrachtet werden.

[2] „Raum“ hier im Sinne des englischen Wortes *space*.

Beispiel die Geschwindigkeit der Recheneinheit auf dem ein Algorithmus ausgeführt, augenscheinlich wird eine große Einwirkung. Aber auch die Programmiersprache und die Qualität des vom Compiler [?] generierten Maschinencodes [?] können sich auf die Laufzeit auswirken (vgl. ebd., S. 58).

Keiner dieser Faktoren bezieht sich jedoch auf die Unterschiede zwischen zwei Algorithmen an sich. Für einen fairen Vergleich zweier Programme welche von zwei verschiedenen, die selbe Aufgabe lösenden Algorithmen abgeleitet sind, müssten alle externen Faktoren, welche einen Einfluss auf die zu messenden Ressourcen haben, gleich sein. Nur unter dieser Bedingung würden sich die, das tatsächlich gewünschte Ergebnis verschleiernden, Faktoren gegenseitig aufheben.

Nachdem die Einhaltung dieser Bedingung oft verunmöglichend schwierig ist, gilt es ein anderes Maß als Stellvertreter für die Laufzeit zu verwenden (vgl. Shaffer, 1997, S. 58–59). Der vorrangige Kandidat für ein solches Ersatzmaß ist die Anzahl von *elementaren Handlungen* beziehungsweise *Schritten*, welche ein Algorithmus für die Lösung einer Aufgabe mit einer bestimmten *Größe* benötigt (vgl. Knuth, 1997, S. 97). Die Größe einer Aufgabe sei hier ein Maß für die Größe der Eingangsdaten (vgl. Aho, Hopcraft und Ullman, 1974, S. 2). Die Größe einer Matrixmultiplikationsaufgabe könnte also beispielsweise die größte Dimension der zu multiplizierenden Matrices sein. Die Größe einer Sortieraufgabe ist üblicherweise die Anzahl der zu sortierenden Elemente.

Für eine genaue Definition einer „elementaren Handlung“ wird in Abschnitt 2.4 ein Modell für eine Maschine, welche einen Algorithmus ausführt, spezifiziert. Das verwendete Modell ist die „Random Access Machine“ (kurz RAM). Im Sinne der Lesbarkeit und Verständlichkeit wird anschließend die Pseudoprogrammiersprache „PseudoPascal“ eingeführt und in Beziehung zur Programmiersprache der RAM gesetzt.

Bevor jedoch die RAM behandelt wird, ist es nützlich die asymptotische Analyse

(Abschnitt 2.3) und die Komplexität von Algorithmen (2.2) zu behandeln.

2.2 Komplexität

Von großem Interesse bei der Analyse eines Algorithmus ist üblicherweise seine Wachstumsrate, also jene Rate, mit der die *Kosten* eines Algorithmus wachsen, während die Größe der Aufgabe wächst (vgl. Shaffer, 1997, S. 61). Hierbei seien die Kosten einer Aufgabe die Ressourcen welche zur Lösung der Aufgabe benötigt werden (vgl. ebd., S. 5).

Die Zeit, die ein Algorithmus für die Lösung einer Aufgabe benötigt, ausgedrückt als Funktion der Größe der Aufgabe, wird *Zeitkomplexität* $f_t(n)$ genannt. Das Grenzverhalten der Komplexität während die Größe wächst, wird *asymptotische Zeitkomplexität* genannt.

Es mag die Annahme bestehen, dass durch scheinbar immer schneller werdende Rechner die Bedeutung effizienter Algorithmen, also Algorithmen mit niedriger Komplexität, sinken würde. Das Gegenteil ist jedoch der Fall. Erstens kann die Geschwindkeit von Rechnern nicht mehr so schnell wachsen wie sie es in den letzten Jahrzehnten getan hat (vgl. Kumar, 2015). Zweitens steigt die Größe der lösbaren Aufgaben mit steigender Rechnergeschwindigkeit – die Komplexität eines Algorithmus bestimmt die Steigerung der Aufgabengröße die durch eine Steigerung der Rechnergeschwindigkeit erreicht werden kann (vgl. Aho, Hopcraft und Ullman, 1974, S. 2).

Angenommen die folgenden fünf Algorithmen $A_1, \ldots, A_5$ (siehe auch Abbildung 2.1) mit ihrerer jeweiligen Zeitkomplexität sind bekannt.

Die Zeitkomplexität sei hier die Anzahl der Zeiteinheiten die benötigt werden um eine Eingabe der Größe n zu verarbeiten. Unter der Annahme, dass eine Zeiteinheit einer Milisekunde entspricht, kann Algorithmus A_1 eine Eingabe der Größe 1000

Algorithmus	Zeitkomplexität
A_1	n
A_2	$n \log n$
A_3	n^2
A_4	n^3
A_5	2^n

Wenn nicht anderweitig angegeben sind alle vorkommenden Logarithmen zur Basis 2.

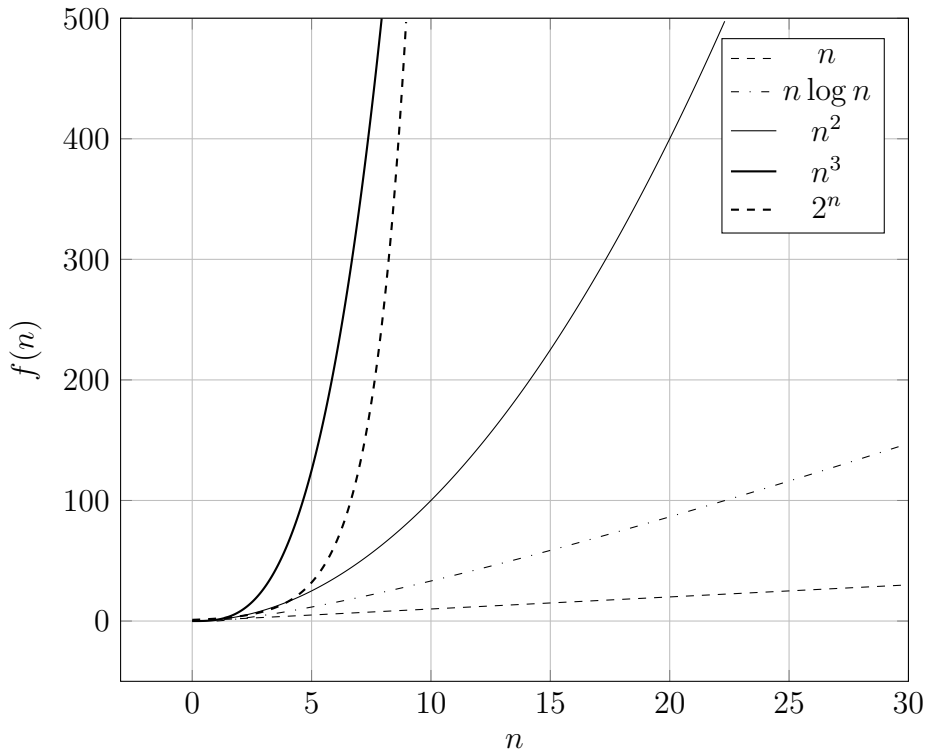

Abbildung 2.1: Komplexität der Algorithmen $A_1, \ldots, A_5$. Die horizontale Achse repräsentiert die Größe. Die vertikale Achse kann die Komplexität bezüglich Zeit, Raum, oder eine andere Ressource repräsentieren – im hier behandelten Beispiel handelt es sich um die Zeitkomplexität.

verarbeiten. Nachdem für Algorithmus A_5 gilt, dass $f_t(n) = 2^n$, gilt für dieses Beispiel also $1000 = 2^n$, womit $n = \log 1000 = 9.996$ bestimmt werden kann. Algorithmus A_5 kann also in der selben Zeit nur Eingaben die höchstens 9 Einheiten groß sind verarbeiten. Tabelle 2.1 gibt eine Übersicht über die Größen der in einer Sekunde, Minute und Stunde durch die fünf gegeben Algorithmen lösbaren Aufgaben.

		Größte Eingabegröße		
Algorithmus	Zeitkomplexität	1 Sek.	1 Min.	1 Stunde
A_1	n	1000	6×10^4	3.6×10^4
A_2	$n \log n$	140	4893	2×10^5
A_3	n^2	31	244	1897
A_4	n^3	10	39	1534
A_5	2^n	9	15	21

Tabelle 2.1: Durch die Wachstumsraten festgelegten größtmögliche lösbare Aufgaben, in verschiedenen Zeitspannen.

Angenommen die nächste Generation von Rechnern ist um ein Zehnfaches schneller als die jetzige Generation. Tabelle 2.2 zeigt die Auswirkung dieser Verschnellerung auf die Größe der lösbaren Probleme. Bei Algorithmus A_5 erhöht eine zehnfache Verschnellerung die Größe der lösbaren Probleme nur um 3, wohingegen sich jene Größe bei Algorithmus A_3 mehr als verdreifacht.

Algorithmus	Größte Eingabegröße jetzige Generation	Größte Eingabegröße „nächste Generation“
A_1	s_1	$10s_1$
A_2	s_2	$\approx 10s_2$ für große s_2
A_3	s_3	$3.16s_3$
A_4	s_4	$2.15s_4$
A_5	s_5	$s_5 + 3.3$

Tabelle 2.2: Auswirkung zehnfach schnellerer Rechner (Aho, Hopcraft und Ullman, 1974, Fig. 1.2).

Statt einer Erhöhung der Geschwindigkeit des Rechners soll nun die Auswir-

kung eines effizienteren Algorithmus untersucht werden. Ausgehend von Tabelle 2.1 und unter Verwendung einer Minute als Vergleichsgrundlage: Das Austauschen von Algorithmus A_4 durch Algorithmus A_3 würde die Lösung Sechsfach größerer Aufgaben erlauben; würde A_4 mit A_2 ausgetauscht werden wären es 125-fach größere. Diese Ergebnisse sind bei weitem imposanter als jene, die durch eine zehnfache Verschnellerung erzielt werden könnten. Daraus lässt sich schließen, dass die asymptotische Komplexität ein wichtiges Merkmal der Güte eines Algorithmus ist (vgl. Aho, Hopcraft und Ullman, 1974, S. 4).

Trotz der Konzentration auf die asymptotische Komplexität ist nicht zu übersehen, dass ein Algorithmus mit einer großen Wachstumsrate eine kleinere Proportionalitätskonstante als ein Algorithmus mit einer niedrigeren Wachstumsrate haben könnte. In dem Fall könnte der schnell wachsende Algorithmus für kleine Aufgabengrößen, möglicherweise sogar für alle Aufgabengrößen die von Interesse sind, überlegen sein. Angenommen die Komplexitäten der Algorithmen $A_1, \ldots, A_5$ wären tatsächlich $1000n$, $100n \log n$, $10n^2$, n^3 und 2^n. In dem Fall wäre A_5 beispielsweise der beste Algorithmus für Probleme der Größe $2 \leq n \leq 9$ (siehe Abbildung 2.2).

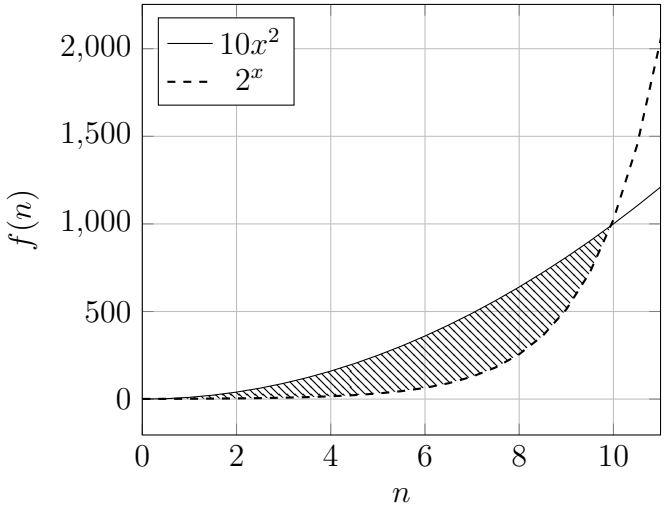

Abbildung 2.2: Komplexität der modifizierten Algorithmen A_5 und A_3. Die horizontale Achse stellt wieder die Größe dar, die vertikale die Komplexität.

2.3 Asymptotische Analyse

> It often happens that we want to evaluate a certain number, defined in a certain way, and that the evaluation involves a very large number of operations so that the direct method is almost prohibitive. In such cases we should be very happy to have an entirely different method for finding information about the number, giving at least some useful approximation to it. And usually this new method gives [...] the better results in proportion to its being more necessary [...]. A situation like this is considered to belong to asymptotics. (Bruijn, 1958, S. 1)

Wie sich noch herausstellen wird ist die genaue Ermittlung der Kosten eines Algorithmus oftmals sehr aufwändig. Aus diesem Grund kann es mitunter hilfreich sein mithilfe der asymptotischen Analyse eine „useful approximation“ dieser Kosten zu ermitteln.

Die zwei nichtnegativen Funktionen $f(n) = n^2$ und $g(n) = n$ kreuzen sich im Punkt $n = 1$. Eine Veränderung der beiden Funktionen zu $f(n) = 2n^2$ und $g(n) = 10n$ verschiebt den Punkt an dem sie sich kreuzen um einen gewissen Wert, aber $f(n)$ übersteigt $g(n)$ noch immer an einer bestimmten Stelle n. Im Allgemeinen verschieben Änderungen an einem konstanten Faktor in beiden Gleichungen nur die *Stelle* an der sich die beiden Linien schneiden, nicht *ob* sie sich schneiden (vgl. Shaffer, 1997, S. 67).

Ein schnellerer Rechner oder ein besserer Compiler [?] verändern die Menge an Aufgaben die durch einen Algorithmus in einer gewissen Zeit lösbar sind um einen konstanten Faktor. Zwei Funktionen die die Zeit, welche ein Algorithmus für eine gewisse Aufgabe benötigt, modellieren werden sich noch immer in einem gewissen Punkt schneiden, ohne Rücksicht auf die Konstanten, welche nun in ihren Laufzeitfunktionen stehen. Aus diesem Grund werden solche Konstanten üblicherweise

ignoriert, wenn eine Schätzung der Laufzeit oder einer anderen Ressource eines Algorithmus für große Aufgabengrößen gefragt ist. Diese Vereinfachung der Analyse wird *asymptotische Analyse* von Algorithmen genannt (vgl. ebd., S. 68).

Ein Werkzeug für diese Vereinfachung ist die O-Notation, die in Unterabschnitt 2.3.1 näher behandelt wird. Die O-Notation wird zur Festlegung von *oberen Grenzen* verwendet (vgl. Knuth, 1997, S. 110). Die Aussage $f(n) = O(g(n))$ (zu lesen als „f von n hat die Ordnung g von n“) sagt aus, dass $g(n)$ eine obere Grenze für den Wert von $f(n)$ bildet, für alle n größer als eine beliebige Konstante. Sie sagt jedoch nichts darüber aus, wie *gut* diese Grenze ist (vgl. Horowitz, Sahni und Rajasekaran, 1997, S. 29). Die folgenden Aussagen sind alle wahr:

$$n = O(n) \qquad n = O(n^2)$$
$$n = O(2^n) \qquad \ldots$$

Um aus einer Aussage der Form $f(n) = O(g(n))$ bedeutungsvolle Informationen ableiten zu können sollte $g(n)$ so klein als möglich sein (vgl. ebd, S. 30). So ist die Aussage $3n + 3 = O(n)$ richtig und oft sinnvoll. Die Aussage $3n + 3 = O(n^2)$ ist zwar genauso richtig, aber deutlich weniger oft bedeutungsträchtig.

Aussagen über „obere Grenzen“ eines Algorithmus, wie auch über seine Komplexität, müssen sich immer auf eine Kategorie von Eingängen der Größe n beziehen (vgl. Horowitz, Sahni und Rajasekaran, 1997, S. 68) – namentlich die günstigsten (*best-case*), ungünstigsten (*worst-case*) und durchschnittlichen (*average* beziehungsweise *expected*[3]) Eingänge (vgl. Shaffer, 1997, S. 63–64). In diesem Kontext spricht man beispielsweise auch von einer *günstigsten Komplexität* (vgl. Aho, Hopcraft und Ullman, 1974, S. 12).

Als Beispiel für die verschienden Kategorien der Eingänge sei der sequenzielle

[3] Aho, Hopcraft und Ullman, 1974, S. 12.

Suchalgorithmus gegeben, der ein Datenfeld mit n Zahlen nach einem bestimmten Wert K durchsucht (vgl. Horowitz, Sahni und Rajasekaran, 1997, S. 63–64). Er beginnt bei der ersten Position im Datenfeld und durchsucht der Reihe nach jedes Feld bis K gefunden wurde, wonach er terminiert. Offensichtlicherweise gibt es viele verschiedene Laufzeiten für diesen Algorithmus: der erste oder letzte Wert des Datenfeldes könnte K sein (*best case* im ersten, *worst case* im zweiten Fall). Meistens werden die Werte jedoch zwischen diesen beiden Stellen im Datenfeld liegen (unter der Annahme, dass das Datenfeld ungeordnet ist). Würde man den Algorithmus, als Programm implementiert, wiederholt nach verschiedenen Werten K im selben Datenfeld suchen lassen so wäre zu erwarten, dass der Algorithmus durchschnittlich etwa $n/2$ Werte durchsucht (*average case*).

2.3.1 O-Notation*

Die Bachmann-Landau O-Darstellung beziehungsweise Notation ersetzt das Wissen über eine Zahl mit gewissen Eigenschaften durch das Wissen, dass eine solche Zahl existiert (vgl. Landau, 1927, S. 3–5).

Angenommen die folgende Information über eine Folge $\{f(n)\}$ ist bekannt:

$$|f(n) - 1| \leq 3n^{-1} \quad (n = 1, 2, 3, \ldots) \tag{2.3.1}$$

Wie oben bereits festgestellt wäre es für die Nützlichkeit von (2.3.1) oft irrelevant wenn die Zahl 3 durch die Zahl 10^8 oder eine andere Konstante ersetzt werden würde. In solchen Fällen ist

$$\begin{cases} \text{Es gibt eine Zahl } A \text{ (unabhängig von } n\text{) derart, dass} \\ \qquad |f(n) - 1| \leq An^{-1} \quad (n = 1, 2, 3, \ldots) \end{cases} \tag{2.3.2}$$

ausreichend. Dies ist der Ansatz der mit dem Symbolismus

$$f(n) - 1 = O(n^{-1}) \quad (n = 1, 2, 3, \ldots) \tag{2.3.3}$$

ausgedrückt wird. Der Ausdruck $O(\phi)$ bedeutet „something that is, in absolute value, at most a constant multiple of the absolute value of $[\phi]$“ (Bruijn, 1958, S. 4). Allgemeiner formuliert gilt, wenn f und g über eine Menge S definiert sind, dann bedeutet die Formel

$$f(s) = O(g(s)) \quad (s \in S) \tag{2.3.4}$$

dass es eine positive ganze Zahl A gibt (die unabhängig von s ist) derart, dass

$$|f(s)| \leq A|g(s)| \quad \text{für alle } s \in S \tag{2.3.5}$$

gilt (vgl. ebd., S. 4).

Einige Beispiele dieser Form sind im Folgenden angegeben (vgl. ebd,, S. 4):

$$\begin{aligned} x^2 &= O(x) \quad (|x| < 2) \\ \sin x &= O(1) \quad (-\infty < x < \infty) \\ \sin x &= (x) \quad (-\infty < x < \infty) \end{aligned}$$

In den obigen Gleichungen wurde immer ein exaktes Intervall angegeben. Eine modifzierte Version der O-Notation, die noch mehr Informationen unterdrückt, ist in (2.3.6) gegeben, wo der Fokus auf großen, positiven Werten von x liegt ($x \to \infty$).

Die Formel

$$f(x) = O(g(x)) \quad (x \to \infty) \tag{2.3.6}$$

bedeutet, dass eine reelle Zahl a existiert derart, dass

$$f(x) = O(g(x) \quad (a < x < \infty) \tag{2.3.7}$$

(vgl. ebd., S. 5). Anders ausgedrückt bedeutet (2.3.6), dass es zwei Zahlen a und A derart gibt, dass

$$|f(x)| \leq A|g(x)| \text{ wenn } a < x < \infty \tag{2.3.8}$$

$$|f(x)| \leq A|g(x)| \text{ wenn } x \geq a \tag{2.3.9}$$

Diese modifizierte Version der O-Notation wird im Rest der Arbeit verwendet werden. (2.3.8) und (2.3.9) sind im Kontext dieser Arbeit gleichwertig, letztere ist die Form wie sie beispielsweise in Knuth, 1997 durchgehend verwendet wird.

Bis jetzt wurde noch nicht definiert was $O(g(s))$ bedeutet, es wurden nur die Bedeutungen einiger vollständigen Formeln erläutert. Tatsächlich kann der alleinstehende Ausdruck $O(g(s))$ nicht definiert werden, zumindest nicht in einer Art und Weise in der (2.3.4) äquivalent zu (2.3.5) bleibt (vgl. ebd., S. 5). Offensichtlicherweise sind $f(s) = O(g(s))$ und $2fs) = O(g(s))$ beide gültige Ausdrücke. Würde $O(g(s))$ eine alleinstehende Bedeutung haben so wäre $f(s) = O(g(s)) = 2f(s)$ und damit $f(s) = 2f(s)$ ableitbar.

Dieses scheinbare Paradoxon tritt wegen eines „Missbrauchs“ des Gleichheitszeichens = auf (vgl. Bruijn, 1958, S. 5). Folgend werden Gleichungen in denen die O-Notation verwendet wird als „einseitig gerichtete Gleichungen“ angesehen, ihre rechte Seite liefert gleich viel Information wie die linke: tatsächlich kann die rechte Seite als eine „Vereinfachung“ der Linken angesehen werden (vgl. Knuth, 1997, S. 108).

Die Verwendung des Zeichens = kann präziser ausgedrückt werden: Formeln die $O(g(s))$ beinhalten können als Mengen von Funktionen von s angesehen werden.

Der Ausdruck $O(g(s))$ steht nun also für die Menge aller ganzzahligen Funktionen f derart, dass zwei Konstanten a und A existerien mit $|f(s)| \leq A|g(s)|$ für alle $s \geq a$. Sind $\alpha(x)$ und $\beta(x)$ Formeln in welchen die O-Notation verwendet wird, so bedeutet der Ausdruck $\alpha(x) = \beta(x)$, dass die von $\alpha(x)$ ausgedrückte Menge von Funktionen in $\beta(x)$ enthalten ist (vgl. Knuth, 1997, S. 108).

2.4 Random Access Machinen*

Eine Random Access Machine modelliert einen Computer mit einem zentralen Rechenregister, in welchem sich Anweisungen nicht selbst modifizieren können. Sie besteht aus einem schreibgeschützten Ausgangsmedium, einem lesegeschützten Eingangsmedium, einem Programm und einem Speicher (siehe Abbildung 2.3).

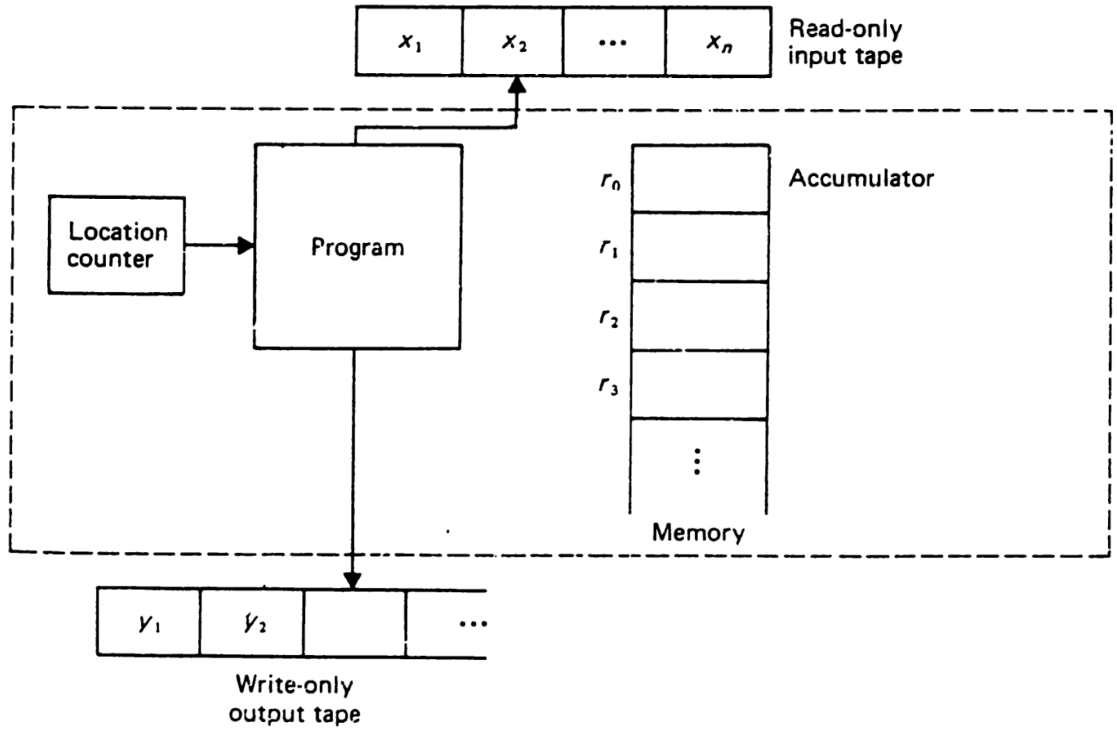

Abbildung 2.3: Eine Random Access Maschine, Abbildung entnommen aus Aho, Hopcraft und Ullman, 1974, Fig. 1.3.

Der Speicher besteht aus einer Folge von Registern $r_0, r_1, \ldots, r_n$ welche ganze Zahlen beliebiger Länge enthalten können (vgl. Aho, Hopcraft und Ullman, 1974, S. 5–6). Der erste Register r_0 wird „Akkumulator“ (*accumulator*) genannt.

Sowohl das Eingangs- als auch das Ausgangsmedium können wie eine Warteschlange betrachtet werden. Wird vom Eingangsmedium gelesen so bewegt sich ein imaginärer, nur in eine Richtung bewegbarer, „Lesekopf“. Einmal gelesene Werte können also nicht wieder gelesen werden. Analog dazu ist auch das Ausgangsmedium zu betrachten.

Das Programm ist eine Folge von (optional mit einer Bezeichnung versehenen) Anweisungen, die nicht im Speicher liegen; daraus folgt, dass sich das Programm nicht selbst modifizieren kann. Die Beschaffenheit der Anweisungen selbst ist nicht übermäßig relevant, sofern sie den auf echten Rechnern zu findenden Anweisungen ähneln (vgl. Aho, Hopcraft und Ullman, 1974, S. 6). Tabelle 2.3 dient als Beispiel für eine solche Menge von Anweisungen und umfasst arithmetische Anweisungen, Eingangs-, und Ausgangsanweisungen und Verzweigungsanweisungen.

Anweisung	Adresse
LOAD	Operand
STORE	Operand
ADD	Operand
SUB	Operand
MULT	Operand
DIV	Operand
READ	Operand
WRITE	Operand
JUMP	Bezeichnung
JGTZ	Bezeichnung
JZERO	Bezeichnung
HALT	

Tabelle 2.3: Tabelle von RAM Anweisungen, entnommen aus Aho, Hopcraft und Ullman, 1974, Fig. 1.4

Ein Operand kann einer der folgenden Ausdrücke sein (vgl. Aho, Hopcraft und Ullman, 1974, S. 7):

1. $= i$, die ganze Zahl i selbst anzeigend.
2. Eine nicht negative ganze Zahl i, den Inhalt des Zählers r_i anzeigend.
3. $*i$, den Inhalt des Zählers r_j anzeigend, wobei j die in Zähler r_i liegende ganze Zahl ist. Gilt $j < 0$ so hält die Maschine an.

Für die Definition der Bedeutung eines Programms seien zwei weitere Größen definiert: eine „speicherabbildende“ Funktion $c(i)$ und einen Befehlszähler, der die nächste auszuführende Anweisung bestimmt. $c(i)$ ist jene ganze Zahl welche im Zähler r_i gespeichert ist (vgl. ebd., S. 7).

Anfangs sind alle Register inklusive des Akkumulators auf Null ($c(i) = 0$ für alle $i \geq 0$) und der Befehlszähler auf die erste Anweisung des Programmes gesetzt, das Ausgabemedium wurde nicht beschrieben. Nach der Ausführung der ersten Anweisung springt der Befehlszähler automatisch auf 2, nach der Ausführung der k-ten auf $k + 1$, außer die erste oder k-te Anweisung ist JUMP, HALT, JGTZ oder JZERO (vgl. ebd., S. 7).

Um die Bedeutung einer Anweisung spezifizieren zu können sei $v(a)$, der Wert des Operands a, wie folgt definiert (vgl. ebd., S.7):

$$
\begin{aligned}
v(= i) &= i, \\
v(i) &= c(i), \\
v(*i) &= c(c(i)).
\end{aligned}
$$

Tabelle 2.4 definiert die Bedeutung der Anweisungen in Tabelle 2.3. Nicht definierte Anweisungen wie STORE $= i$, oder nicht definierte Operationen wie eine Division durch Null, können als äquivalent zu HALT angesehen werden (vgl. ebd., S.7.).

Anweisung	Bedeutung
LOAD a	$c(0) \leftarrow v(a)$
STORE $i/*i$	$c(i) \leftarrow c(0)$ bzw. $c(c(i)) \leftarrow \ldots$
ADD a	$c(0) \leftarrow c(0) + v(a)$
SUB a	$c(0) \leftarrow c(0) - v(a)$
MULT a	$c(0) \leftarrow c(0) \times v(a)$
DIV a	$c(0) \leftarrow \lfloor c(0)/v(a) \rfloor$
READ $i/*i$	$c(i)$ bzw. $c(c(i)) \leftarrow$ gegenwärtiger Eingabewert. Der vorgestellte „Lesekopf“ bewegt sich um eine Einheit.
WRITE a	$v(a)$ wird ausgegeben. Der analog zum „Lesekopf“ vorgestellte „Schreibkopf“ bewegt sich um eine Einheit.
JUMP b	Der Befehlszähler wird auf die Anweisung mit der Bezeichnung b gesetzt.
JGTZ b	Gilt $c(0) > 0$ so ist dieser Ausdruck äquivalent zu JUMP b, andernfalls wird der Befehlszähler um eins erhöht.
JZERO b	Gilt $c(0) = 0$ so ist dieser Ausdruck äquivalent zu JUMP b, andernfalls wird der Befehlszähler um eins erhöht.
HALT	Die Ausführung endet.

Tabelle 2.4: Bedeutung der RAM Anweisungen aus Tabelle 2.3, vgl. Aho, Hopcraft und Ullman, 1974, Fig. 1.5. Der Operand a steht für $= i$, i oder $*i$.

Während der Ausführung der ersten acht in Tabellen 2.3 und 2.4 aufgelisteten Anweisungen (LOAD bis WRITE, in letzterer Tabelle durch eine Linie gekennzeichnet) wird der Befehlszähler um eins erhöht. Dementsprechend werden die Anweisungen eines Programmes in aufeinanderfolgender, sequenzieller Reihenfolge ausgeführt bis eine der übrigen vier Anweisungen (JUMP, JGTZ, JZERO und HALT) vorgefunden wird[4] (vgl. ebd., S. 7).

Um die Zeit welche jede dieser Anweisungen benötigt genau spezifizieren zu können existieren zwei Modelle: das *einheitliche*, und das *logarithmische* Kostenmodell (vgl.Aho, Hopcraft und Ullman, 1974, S. 12). In ersterem benötigt jede RAM Anweisung genau eine Zeiteinheit. In letzterem Modell spielt die Anzahl der im Speicher benötigten Bits eine Rolle.

So beschreibt

$$l(i) = \begin{cases} \lfloor log|i| + 1 \rfloor & i \neq 0 \\ 1 & i = 0 \end{cases}$$

die Anzahl von Bits die für das speichern der Zahl i benötigt werden. Beispielsweise bringt nun der Operand $= i$ die Kosten $l(i)$ mit sich, der Operand i die Kosten $l(i) + l(c(i))$ (vgl. ebd., S.12).

2.5 PseudoPascal*

Obwohl der in dieser Arbeit verwendete grundlegende Maßstab für die Komplexität eines Algorithmus Bezug auf die Anweisungen einer RAM nimmt, werden Algorithmen im Folgenden nicht in Form einer solchen primitiven Maschine beschrieben. Neben

[4]Bei JGTZ und JZERO ist für die Gültigkeit dieser Aussage anzunehemen, dass jeweils $c(0) \leq 0$ oder $c(0) \neq 0$ gilt.

der Beschreibung durch die natürliche Sprache Deutsch, wie sie in Abschnitt 1.1 eingeführt wurde, werden Algorithmen auch durch die höhere Programmiersprache „PseudoPascal“[5] dargestellt.

Ein in PseudoPascal geschriebenes Programm kann ohne weiteres in ein RAM-Programm übersetzt werden, tatsächlich wäre genau das die Rolle eines Compilers für PseudoPascal. Dieser Unterabschnitt soll jedoch nicht als Vorlage für einen hypothetischen Compiler dienen, die Details der Übersetzung werden nur oberflächlich behandelt. Für den Zweck dieser Arbeit ist es ausreichend, nur die benötigte Zeit und den benötigten Raum eines übersetzten RAM-Ausdrucks, der zu einem Ausdruck in PseudoPascal gehört, zu behandeln.

PseudoPascal erlaubt die Verwendung jeglicher mathematischer Aussagen, sofern ihre Bedeutung klar ist und die Übersetzung zu RAM-Anweisungen offenkundig ist. Gleichermaßen hat die Sprache keine festgelegten Datentypen: Variablen können Zahlen, Zeichenketten, Folgen und je nach Bedarf andere Datentypen darstellen. Der Datentyp einer Variable und ihr Geltungsbereich[6] sollte durch ihren Namen oder dem Kontext in dem sie erscheint offensichtlich sein.

Ein PseudoPascal-Programm setzt sich aus den folgenden Anweisungen zusammen:

1. Variable := Ausdruck
2. **if** Bedingung **then** Anweisung [**else** Anweisung][7]
3. (a) **while** Bedingung **do** Anweisung
   (b) **repeat** Anweisung **until** Bedingung
4. **for** Variable := Anfangswert **until** Endwert **do** Anweisung

[5] Die Definition der Sprache „PseudoPascal“ ist angelehnt an „Pidgin ALGOL“ welches in Aho, Hopcraft und Ullman, 1974, S. 33–39 definiert wird.

[6] Der *Geltungsbereich* einer Variable ist das Umfeld beziehungsweise der Bereich in dem sie eine Bedeutung hat (vgl. Aho, Hopcraft und Ullman, 1974, S. 34).

[7] „**else** Anweisung“ ist optional. Ein **else** wird dem nächstgelegendem unzugeteilten **then** zugeordnet.

5. **begin**
   Anweisung
   [Anweisung
   ...
   Anweisung]
   end
6. (a) **function** Name (Parameterliste): Anweisung
   (b) return Ausdruck
   (c) Name (Parameter)
7. (a) **read** Variable
   (b) **write** Ausdruck
8. {*Kommentar* }

Zusammenfassend zur Funktionsweise dieser Anweisungen:

1. Die Zuweisungsanweisung

   Variable := Ausdruck

entspricht der Zuweisung aus Abschnitt 1.1. Der Ausdruck rechts des Zuweisungsoperators „:=“ wird ausgewertet und der resultierende Wert der links stehenden Variable zugewiesen. Der Zeitaufwand der Zuweisungsanweisung ist die Zeit, die benötigt wird um den Ausdruck auszuwerten und die Zuweisung durchzuführen. Üblicherweise wird diese Anweisung eine Zeiteinheit kosten.

2. In der **if** Anweisung

   if Bedingung **then** Anweisung [**else** Anweisung]

kann die Bedingung welche auf das **if** folgt, ein beliebiger Ausdruck der den Wert **true** oder **false** hat, sein. Ein solcher Ausdruck kann die Form Variable = Variable,

Variable = Konstante oder Variable = Ausdruck annehmen, wobei die Ausdrücke auf beiden Seiten des Gleichheitszeichens vertauscht werden können. Ist die Bedingung wahr, so wird die auf **then** folgende Anweisung ausgeführt. Andernfalls wird die auf **else** folgende Anweisung ausgeführt, sofern angegeben. Die Kosten der **if** Anweisung sind in Summe die Kosten für das Auswerten und Prüfen des Ausdrucks, beziehungsweise der Bedingung, zuzüglich der Kosten des Ausdrucks welcher auf **then** oder **else** folgt, welche auch immer tatsächlich ausgeführt wird.

3. Der Zweck der **while** Anweisung

while Bedingung **do** Anweisung

und der **repeat** Anweisung

repeat Anweisung **until** Bedingung

ist das Erzeugen einer Schleife – sie werden auch **while**- und **repeat** Schleife genannt. In der **while** Anweisung wird die auf das **while** folgende Bedingung ausgewertet. Ist sie **true**, so wird die auf das **do** folgende Anweisung ausgeführt. Dieser Vorgang wird wiederholt bis die Bedingung **false** wird. Wieder sind die Kosten der **while** Anweisung in Summe die Kosten der Auswertung und Prüfung der Bedingung zuzüglich der Kosten für das Ausführen der Anweisung, so oft sie ausgeführt wird.

Die **repeat** Anweisung ist ähnlich, mit der Ausnahme, dass die Aussage welche auf **repeat** folgt vor der auszuwertenden Bedingung ausgeführt wird.

4. In der **for** Anweisung (auch: **for** Schleife)

for Variable := Anfangswert **until** Endwert **do** Anweisung

sind Anfangswert und Endwert Ausdrücke. Die auf das **do** folgende Anweisung wird ausgeführt und der Wert der Variable (in diesem Kontext *Index* genannt) wird um eins erhöht und mit dem Endwert verglichen. Dieser Vorgang wird wiederholt bis

der Index größer als der Endwert ist. Die Kosten dieser Anweisung sind aus jenen der **while** Schleife abzuleiten.

5. Eine Folge von Anweisungen, die von den Schlagworten **begin** und **end** umgeben ist, wird *Block* genannt. Er kann demnach überall verwendet werden wo eine Anweisung erforderlich ist: So können beispielsweise mehrere Anweisungen innerhalb einer **for** Schleife ausgeführt werden:

for i := 0 **until** 9 **do**
begin
    { *Hier können mehrere Anweisungen, die alle zehn Mal ausgeführt werden, erscheinen* }
end

Üblicherweise werden Programme ebenfalls ein *Block* sein.

6. *Functions* beziehungsweise „Funktionen“ werden durch die Anweisung

function Name (Parameterliste): Anweisung

definiert. Beispielsweise definiert die folgende Anweisung eine Funktion, genannt „min“.

function min(x, y):
if x > y **then** return y **else** return x

Die return Anweisung terminiert eine Funktion nachdem der Wert des auf das return folgenden Ausdrucks ermittelt wurde. Der Wert der Funktion ist der Wert dieses Ausdrucks. Ein Aufruf der Funktion min würde nun wie folgt erfolgen.

a := 5
b := 10
c := min(a, b)
{ *c hat nun den Wert '5'* }

Funktionen die keinen Rückgabewert besitzen, die also keine return Anweisung aufweisen, werden Prozeduren genannt. Eine Prozedur interchange(x, y) die entsprechend definiert ist, wird wie folgt aufgerufen.

$$\text{interchange}(a_i,\, a_{i+1})$$

Die

7. Offensichtliche Bedeutung haben die **read** und **write** Anweisungen. Die Kosten von **read** sind (immer) eins. Die Kosten von **write** sind eins plus die Kosten der Auswertung des auf das Schlagwort **write** folgenden Ausdrucks.

8. Kommentare wie

$$\{\ \textit{Erhöhe die Variable um eins}\ \}$$
$$\text{a} := \text{a} + 1$$

haben keine Kosten und erlauben das Einfügen von Bemerkungen. Sie werden überwiegend zur Erläuterung von wichtigen Stellen in Programmen verwendet.

Diese Beschreibung von PseudoPascal ist allerdings in keinster Weise lückenlos definiert: Es besteht durchaus die Möglichkeit, Programme zu schreiben, deren Bedeutung von Details abhängt die nicht abgedeckt wurden. Die im Folgenden noch dargestellten Programme haben keine solche Abhängigkeiten und bewegen sich nur im wohldefinierten Spektrum.

Zur Demonstration der Übersetzung eines PseudoPascal Programms in ein RAM Programm sei die Funktion $f(n)$

$$f(n) = \begin{cases} n^n & \text{für alle ganze Zahlen } n \geq 1 \\ 0 & \text{andernfalls.} \end{cases}$$

zu betrachten. Das PseudoPascal Programm 2.1 berechnet $f(n)$ indem es $n-1$ Male n mit sich selbst multipliziert. Ein entsprechendes RAM Programm ist in 2.5 gegeben, wobei die Variablen $r1, r2$ und $r3$ jeweils in den Zählern 1, 2, und 3 gespeichert sind.

```
1  begin
2      read r1
3      if r1 ≤ 0 then
4          write 0
5      else begin
6          r2 := r1
7          r3 := r1 − 1
8          while r3 > 0 do begin
9              r2 := r2 ∗ r1
10             r3 := r3 − 1
11         end
12         write r2
13     end
14 end
```

Programm 2.1: PseudoPascal Programm für n^n

	RAM Programm		Entsprechende PseudoPascal Anweisungen
	READ	1	**read** r1
	LOAD JGTZ WRITE	1 pos =0	**if** $r1 \leq 0$ **then write** 0
	JUMP	endif	
pos:	LOAD STORE	1 2	r2 := r1
	LOAD SUB STORE	1 =1 3	r3 := r1 − 1
while:	LOAD JGTZ JUMP	3 continue endwhile	**while** $r3 > 0$ **do**
continue:	LOAD MULT STORE	2 1 2	r2 := $r2 \times r1$
	LOAD SUB STORE	3 =1 3	r3 := $r3 - 1$
	JUMP	while	
endwhile: endif:	WRITE HALT	2	**write** r2

Tabelle 2.5: RAM Programm für n^n mit jeweilig entsprechenden PseudoPascal Anweisungen.

3 Sortieren

Im Duden ist „sortieren“ definiert als „nach Art, Farbe, Größe, Qualität o. Ä. sondern, ordnen“ (Dudenredaktion, 2020). Die hier verwendete Definition des Wortes ist deutlich enger, sie beschränkt sich auf das Arrangieren von Elementen in aufsteigender Ordnung.

Mathematisch ausgedrückt ist also eine Folge von n Zahlen (a_1, a_2, …, a_n[1].) gegeben, von der eine Permutation $(a'_1, a'_2, \ldots, a'_n)$ zu finden ist für die gilt, dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$ (vgl. Cormen u. a., 2001, S. 123).

Im folgenden Kapitel wird anhand des „Insertionsort“ Algorithmus die konkrete Analyse von Sortieralgorithmen dargestellt. Dieser Algorithmus ist einer von Hunderten die alle verschiedene Vor- und Nachteile haben. Es gibt jedoch nicht einen (bekannten) *besten* Weg etwas zu sortieren – es gibt, abhängig von den zu sortierenden Daten, dem verwendeten Rechner, et cetera, viele beste Wege (vgl. Knuth, 1998, S. 74).

3.1 Insertionsort

Die „Sortierung durch Einfügung“ (*sorting by insertion*) ist eine der wichtigsten Arten der Sortierung. Bevor Eintrag a_j geprüft wird, wird angenommen, dass die vorangehenden Einträge $a_1, \ldots, a_{j-1}$ bereits sortiert sind; dann wird a_j an dem

[1] Folgend wird diese Folge auch nur als $a_1, \ldots, a_n$ angeschrieben

passenden Platz in der vorangehenden, sortierten Teilliste eingefügt.

Algorithmus P (*Purer Insertionsort*). Die Zahlen $a_1, \ldots, a_n$ werden geordnet. Nach der Termination des Algorithmus werden sie die Ordnung $a_1 \leq \cdots \leq a_n$ haben. Zur Algorithmusbeschreibung vgl. Knuth, 1998, S. 81.

P1 [Iteriere über j.] Führe die Schritte P2 bis einschließlich P5 für $j = 2, 3, \ldots, n$ aus. Terminiere anschließend den Algorithmus.

P2 [Lege i, und A an.] Setze $i \leftarrow j - 1$ und $A \leftarrow a_j$. (In den folgenden Schritten wird versucht werden, durch vergleichen von A mit a_i für absteigende Werte von i, A in die richtige Position zu bringen.)

P3 [Vergleiche A mit a_i.] Gilt $A \leq a_i$, so gehe zu Schritt P5. (Die richtige Position für den Eintrag A ist gefunden worden.)

P4 [Verschiebe a_i, verringere i.] Setze $a_{i+1} \leftarrow a_i$, dann $i \leftarrow i - 1$. Gilt $i > 0$, so gehe zurück zu Schritt P1. (Wenn gilt, dass $i = 0$, dann ist A die bis jetzt kleinste gefundene Zahl. Folglich muss A an die erste Stelle.)

P5 [a_{i+1} zu A.] Setze $a_{i+1} \leftarrow A$.

Eine Implementation von Algorithmus P ist in 3.1 gegeben.

```
1  begin
2      for j := 2 to n do begin
3          A := a_j
4          i := j − 1
5          while i > 0 and a_i > A do begin
6              a_{i+1} := a_i
7              i := i − 1
8          end
9          a_{i+1} := A
10     end
```

11 **end**

Programm 3.1: Purer Insertionsort

Tabelle 3.1 gibt eine Übersicht über die Laufzeiten der Anweisungen von Programm 3.1, unter der stark vereinfachenden Annahme, dass jede Zeile gleich und konstant viel Zeit kostet.

	Programm 3.1	Kosten	Anzahl
1	**begin**	0	1
2	**for** j := 2 **to** n **do begin**	c_2	n
3	A := a_j	c_3	$n-1$
4	i := j − 1	c_4	$n-1$
5	**while** $i > 0$ **and** $a_i > key$ **do begin**	c_5	$\sum_{j=2}^{n} t_j$
6	$a_{i+1} := a_i$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7	i := i − 1	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8	**end**	0	$\sum_{j=2}^{n}(t_j - 1)$
9	a_{i+1} := A	c_9	$n-1$
10	**end**	0	$n-1$
11	**end**	0	1

Tabelle 3.1: Kosten und Anzahl der Vorkommnisse der einzelnen Anweisungen von Programm 3.1

Die Laufzeit eines Algorithmus ist die Summe der Laufzeiten der einzelnen ausgeführten Anweisungen; eine Anweisung die c Schritte für ihre Ausführung benötigt und n Mal ausgeführt wird trägt cn Zeiteinheiten zur gesamten Laufzeit bei (vgl. Cormen u. a., 2001, S. 24). Um die Zeitkomplexität $f_t(n)$ des Algorithmus zu errechnen werden in 3.1.1 die Produkte der Spalten „Kosten“ und „Anzahl“ summiert, wobei n die Anzahl der zu sortierenden Elemente ist.

$$f_t(n) = c_2 n + c_3(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_9(n-1) \quad (3.1.1)$$

Im Folgenden beschreibt t_j für alle $j = 2, 3, \ldots, n$ wie oft die **while** Schleife in Zeile 5 für einen gewissen Wert j ausgeführt wird.

Der günstigste Fall für diesen Algorithmus ist eine bereits sortierte Eingabeliste (vgl. Cormen u. a., 2001, S. 25). Für alle $j = 2, 3, \ldots, n$ wird in Zeile 5 ermittelt, dass $a_i \leq A$ wenn i den Initialwert $j - 1$ hat. Demzufolge gilt $t_j = 1$ für $j = 2, 3, \ldots n$.

$$f_t(n) = c_2 n + c_3(n-1) + c_4(n-1) + c_5(n-1) + c_9(n-1)$$

was zu

$$f_t(n) = (c_2 + c_3 + c_4 + c_5 + c_9)n - (c_3 - c_4 - c_5 - c_9) \tag{3.1.2}$$

vereinfacht werden kann. 3.1.2 hat die Form $an + b$ wobei a und b Konstanten sind – sie ist also eine *lineare Funktion* von n, die Zeitkomplexität ist im günstigsten Fall $O(n)$.

Sind die Elemente in umgekehrt sortierter Reihenfolge, sind in diesem Fall also die Zahlen absteigend sortiert, so tritt der ungünstigste Fall ein. Jedes Element a_j muss mit jedem Element der sortierten Unterliste $a_1, \ldots, a_j - 1$ verglichen werden. Somit ist also $t_j = j$ für die bekannte Folge von j.

Nachdem

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

und

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

(vgl. Knuth, 1997, S. 32) ergibt sich die ungünstigste Zeitkomplexität

$$f_t(n) = c_2 n + c_3(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_9(n-1)$$

die zu

$$f_t(n) = \left(\frac{c_5 + c_6 + c_7}{2}\right) n^2 + \left(c_2 + c_3 + c_4 + \frac{c5 + c_6 + c_7}{2} + c_9\right) - (c_3 + c_4 + c_5 + c_9) \quad (3.1.3)$$

zusammengefasst werden kann. 3.1.3 hat die Form $an^2 + bn + c$ wobei a, b und c Konstanten sind, sie ist eine *quadratische Funktion* von n – die Zeitkomplexität des Algorithmus ist also im ungünstigsten Fall $O(n^2)$.

Wie zu Beginn dieses Abschnitts bereits erwähnt, wurde in diesem Prozess zur Ermittlung der Zeitkomplexität eine Vereinfachung vorgenommen: Die Annahme, dass jede Zeile des Programms gleich viel Zeit kostet, und dass sich diese Kosten im Laufe des Programms nicht verändern. Eine detailliertere Analyse könnte beispielsweise mit dem in Abschnitt 2.4 dargestellten Modell der Random Access Machine vollzogen werden. So könnten die Kosten für die jeweiligen Zeilen beziehungsweise Anweisungen mithilfe des einheitlichen oder logarithmischen Kostenmodells errechnet und anstatt der jeweiligen Konstante c_{Zeile} eingesetzt werden. Dies fand hier nicht statt und wird auch im Folgenden nicht vorgenommen werden, eine solche Analyse würde den Rahmen dieser Arbeit sprengen.

4 Suchen

Dieses Kapitel beschäftigt sich mit der Suche eines bestimmten Elements A in einer Folge von Zahlen $a_1, \ldots, a_n$.

Such- und Sortiervorgänge hängen oft eng miteinander zusammen. Als Beispiel ist folgendens Problem zu betrachten: Zwei Mengen von Zahlen, $A = \{a_1, a_2, \ldots, a_m\}$ und $B = \{b_1, b_2, \ldots, b_n\}$, sind gegeben; ist $a \subseteq B$? Zwei Lösungswege sind offenkundig (vgl. Knuth, 1998, S. 394):

1. Vergleiche jedes a_i mit jedem b_j bis ein Treffer gefunden wurde.
2. Sortiere die as und bs, und gehe, die passende Bedingung prüfend, sequentiell durch die beiden Folgen.

Beide dieser Wege sind für gewisse Wertebereiche von m und n angebracht. Lösung 1 ist für kleine m und n zu bevorzugen, Lösung 2 ist für große n bedeutend schneller.

Analog zu den oben dargestellten Lösungswegen beschäftigt sich Abschnitt 4.1 mit der Suche in einer unsortierten Liste und 4.2 mit der Suche in einer bereits sortierten Liste.

4.1 Sequentielle Suche

„Begin at the beginning,and go on till you find the right key; then stop.“(Knuth, 1998, S. 396). Dieser Algorithmus kann wie folgt präziser formuliert werden:

Algorithmus S (*Sequentielle Suche*). Die Zahlen $a_1, \ldots, a_n$ werden nach einer Zahl A durchsucht. $n \geq 1$ wird angenommen.

S1 [Bereite vor.] Setze $i \leftarrow 1$.

S2 [Vergleiche.] Gilt $A = a_i$ so terminiert der Algorithmus mit Erfolg.

S3 [Gehe forwärts.] Erhöhe i um 1.

S4 [Ende?] Gilt $i \leq n$, so gehe zurürck zu Schritt S2. Andernfalls terminiert der Algorithmus ohne Erfolg.

Zu bemerken ist, dass der Algorithmus auf zwei verschiedene Arten terminieren kann, *mit Erfolg* (wurde die gewünschte Zahl gefunden) oder *ohne Erfolg* (wurde festgestellt, dass die gewünschte Zahl nicht vorhanden ist). Selbiges gilt für fast alle Suchalgorithmen (vgl. Knuth, 1998, S. 396).

Die sequentielle Suche benötigt im ungünstigsten Fall anschaulicherweise n Zeiteinheiten, Algorithmus S ist also der Ordnung $O(n)$.

4.2 Suche durch Schlüsselvergleiche

Diese Art der Suche basiert auf einer linearen Sortierung der zu durchsuchenden Folge. Wird nun der zu findende Eintrag A mit einem Eintrag a_i verglichen kann die Suche auf drei Arten fortgesetzt werden, abhängig davon, ob $A < a_i$, $A = a_i$ oder $A > a_i$ gilt. Der sequentiellen Suche wie sie in 4.1 behandelt wird stehen dagegen nur die Vergleiche $A = a_i$ und $A \neq a_i$ zur Verfügung (vgl. Knuth, 1998, S. 409).

Ist eine große Datenmenge zu sortieren ist das sequentielle Durchsuchen oft keine Option (vergleichbar mit dem Suchen eines Namens anhand einer Telefonnummer in einem Telefonbuch), sind die Daten jedoch sortiert ist der Vorgang um ein vielfaches weniger aufwändig (vergleichbar mit dem Suchen einer Telefonnummer anhand eines Namens).

Wenn eine Datenmenge nur einmal zu durchsuchen ist, wäre eine sequentielle

Suche natürlich schneller als die gesamten Daten zuerst zu sortieren; werden jedoch wiederholte Durchsuchungen benötigt so ist eine sortierte Datenmenge von großem Vorteil. Demzufolge liegt in diesem Abschnitt der Fokus auf der Suche in einer Liste für dessen Einträge

$$a_1 < a_2 < \cdots \leq a_n$$

gilt. Nach einem Vergleich von A und a_i gilt nun entweder:

$$\begin{array}{lll} & A < a_i & a_i, a_{i+1}, \ldots, a_n \text{ werden nicht mehr betrachtet;} \\ \text{oder} & A = a_i & \text{die Suche ist fertig;} \\ \text{oder} & A > a_i & a_1, a_2, \ldots, a_i \text{ werden nicht mehr betrachtet.} \end{array}$$

Algorithmus B (*Binary search*). Die Zahlen $a_1, \ldots, a_n$ sind in aufsteigender Reihenfolge $a_1 < a_2 < \cdots < a_n$ sortiert und werden nache einer Zahl A durchsucht.

B1 [Bereite vor.] Setze $l \leftarrow 1$ und $u \leftarrow n$.

B2 [Ermittle den Mittelwert.] (Hier ist bereits bekannt, dass wenn A vorkommt $a_l \leq A \leq a_u$ gilt.) Gilt $u < l$ terminiert der Algorithmus ohne Erfolg. Andernfalls setze i auf den ungefähren Mittelwert der relevanten Teilliste $i \leftarrow \lfloor (l + u)/2 \rfloor$.

B3 [Vergleiche.] Gilt $A < a_i$, gehe zu Schritt B4. Gilt $A > a_i$ gehe zu Schritt B5. Andernfalls gilt $A = a_i$, der Algorithmus terminiert mit Erfolg.

B4 [Passe u an.] Setze $u \leftarrow i - 1$ und gehe zurück zu B2.

B5 [Passe l an.] Setze $l \leftarrow i + 1$ und gehe zurück zu B2.

In Programm 4.1 ist eine mögliche Implementation von Algorithmus B gegeben.

```
1 function bsearch(A, a)
```

```
2  begin
3      l := 1
4      u := n
5
6      while l ≥ u do begin
7          i := ⌊(l + u)/2⌋ /2
8          if A < a_i then
9              u := i − 1
10         else if A > a_i then
11             l := i + 1
12         else return i
13     end
14 end
```

Programm 4.1: Binary Search Implementation, die Stelle von A in der Liste a wird zurückgegeben

Tabelle 4.1 bietet ein Beispiel zur Funktionsweise von Algorithmus B anhand von acht sortierten Zahlen.

1	2	3	4	5	6	7	8	l	u	i
061	154	275	*509*	612	677	765	908	1	8	4
				612	*677*	765	908	5	8	6
				612				5	5	5

Tabelle 4.1: Beispiel des binären Suchalgorithmus, die Zahl 612 ist gesucht. Die Spalten l, u und i korrespondieren mit den gleichnamigen Variablen in der Algorithmusdefinition.

Um den Binary Sort zu verstehen ist es hilfreich den Vorgang als einen binären Entscheidungsbaum darzustellen, wie es in Abbildung 4.1 für den Fall $n = 16$ geschieht.

Ist $n = 16$ so ist der erste Vergleich des Algorithmus zwischen A und a_8, wie es mit dem Wurzelknoten ⑧ in der Abbildung dargestellt wird. Ist $A < a_8$ folgt der

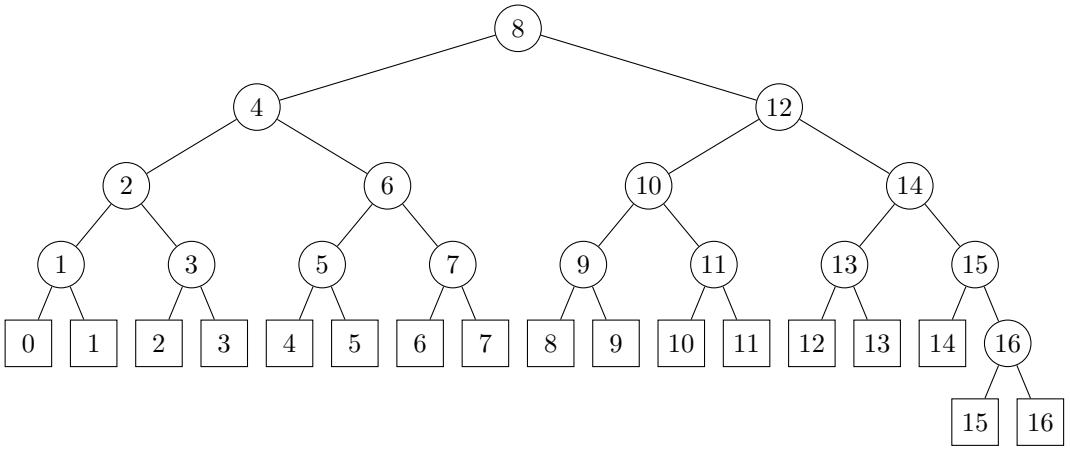

Abbildung 4.1: Ein *comparison tree* („Entscheidungsbaum“) der die Vergleiche die ein Binary Search Algorithmus bei $n = 16$ treffen kann, modelliert (vgl. Knuth, 1998, S. 412)

Algorithmus dem linken Teilbaum, nun A mit a_4 vergleichend. Entsprechend wird wenn $A > a_8$ der rechte Teilbaum verwendet. Eine erfolglose Suche führt zu einer der äußeren rechteckigen Knoten $\boxed{0}$ bis $\boxed{\text{n}}$; beispielsweise wird der Knoten $\boxed{6}$ nur erreicht wenn $a_6 < A < a_7$.

Sind n Elemente zu durchsuchen und gilt $n \neq 0$ so kann ein Binärbaum für Algorithmus B nach den folgenden Regeln aufgebaut werden: Der Wurzelknoten ist $\lceil n/2 \rceil$, der linke Teilbaum ist ein korrespondierende Binärbaum mit $\lceil n/2 \rceil - 1$ Knoten, der rechte ein Binärbaum mit $\lfloor n/2 \rfloor$ Knoten (vgl. Knuth, 1998, S. 412). Werden mit Algorithmus B n Elemente durchsucht hat der korrespondierende Binärbaum also n Knoten. Dies ist auch in Abbildung 4.1 zu sehen, wenn nur die „Entscheidungsknoten“ ($\textcircled{1}$ bis $\textcircled{\text{n}}$) betrachtet werden.

Ist a_{10} zu finden so macht der Algorithmus die folgenden drei Vergleiche: $A > a_8$, $A < a_{12}$ und $A = a_{10}$. In Abbildung 4.1 entspricht dies dem Weg vom Wurzelknoten zu $\textcircled{10}$. Gleichermaßen entspricht das Verhalten von Algorithmus B für andere zu findende Elemente A immer einem Weg der vom Wurzelknoten ausgeht (vgl. Cormen u. a., 2001, S. 257).

Die Tiefe eines Binärbaums mit n Knoten ist $\lfloor \log n \rfloor + 1$ (vgl. Horowitz, Sahni und

Rajasekaran, 1997, S. 79). Nachdem der ungünstigste Fall für Algorithmus B entritt, wenn einer der tiefstliegenden runden Knoten erreicht wird ist der Algorithmus der Ordnung $O(\log n)$.

Anhang

Literatur

Aho, Alfred V., John E. Hopcraft und Jeffrey D. Ullman (1974). *The Design and Analysis of Computer Algorithms*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201000296 (siehe S. 10, 11, 13, 14, 16, 20–25).

Bruijn, Nicolaas G. de (1958). *Asymptotic Methods in Analysis*. Amsterdam: North-Holland Publishing Co. (siehe S. 9, 15, 18, 19).

Callahan, David und John Casey (2015). *Euclid's Ëlements"Redux*. Lulu Enterprises Incorporated. ISBN: 9781312110786 (siehe S. 4).

Cormen, Thomas H. u. a. (2001). *Introduction to Algorithms*. 2nd. Cambridge, Massachusetts: The MIT Press (siehe S. 9, 32, 34, 35, 41).

Dudenredaktion (2020). *Sortieren auf Duden online*. URL: `https://www.duden.de/node/169115/revision/169151` (besucht am 24. 02. 2020) (siehe S. 32).

Gericke, Helmuth (1984). *Mathematik in Antike und Orient*. Springer-Verlag Berlin Heidelberg. ISBN: 978-3-642-68631-3 (siehe S. 3).

Horowitz, Ellis, Sartaj Sahni und Sanguthevar Rajasekaran (1997). *Computer Algorithms*. New York: Computer Science Press (siehe S. 3, 4, 7, 8, 16, 17, 41).

Johnson, David S. (1999). „A theoretician's guide to the experimental analysis of algorithms". In: *Data Structures, Near Neighbor Searches, and Methodology* (siehe S. 9).

Knuth, Donald E. (1997). *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Third. Addison-Wesley (siehe S. 3, 6–10, 16, 19, 20, 36).

Knuth, Donald E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*. Second. Addison-Wesley (siehe S. 32, 33, 37, 38, 41).

Kumar, Suhas (2015). *Fundamental Limits to Moore's Law*. eprint: 1511.05956 (cond-mat.mes-hall) (siehe S. 11).

Landau, Edmund (1927). *Vorlesungen über Zahlentheorie: Bd. Aus der analytischen und geometrischen Zahlentheorie*. Bd. 2. Vorlesungen über Zahlentheorie. Leipzig: S. Hirzel (siehe S. 17).

Melani, Alessandra u. a. (2017). „Exact Response Time Analysis for Fixed Priority Memory-Processor Co-Scheduling". In: *IEEE Transactions on Computers* 66.4, S. 631–646. ISSN: 0018-9340. DOI: 10.1109/TC.2016.2614819 (siehe S. 9).

Pickover, Clifford A. (2009). *The Math Book: From Pythagoras to the 57th Dimension, 250 Milestones in the History of Mathematics*. Sterling Milestones Series. Sterling. ISBN: 9781402757969 (siehe S. 3).

Shaffer, Clifford A. (1997). *A practical introduction to data structures and algorithm analysis*. Prentice Hall Upper Saddle River, NJ (siehe S. 8–11, 15, 16).

Wolff, Christian von (1747). *Vollständisches Mathematisches Lexicon*. Leipzig: Gleditsch (siehe S. 3).

Abbildungsverzeichnis

Tabellenverzeichnis

Eidesstattliche Erklärung

Ich, Laurenz Weixlbaumer, erkläre hiermit eidesstattlich, dass ich diese vorwissenschaftliche Arbeit selbständig und ohne Hilfe Dritter verfasst habe. Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als Zitate kenntlich gemacht und alle verwendeten Quellen angegeben habe.

Linz, am _____ _____

Datum Unterschrift