Effizientes Sortieren - Ein ungelöstes Problem der Informatik?

Christoph Deiser

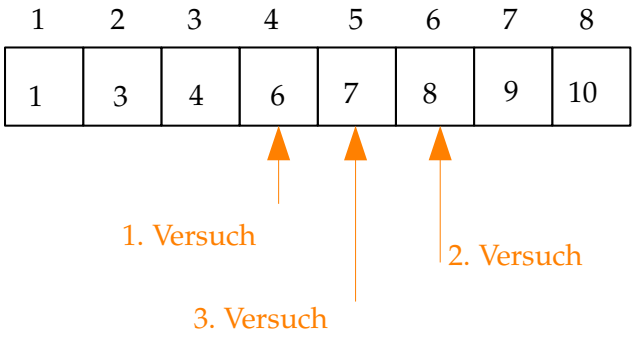
Zusammenfassung—Der optimale Sortieralgorithmus ist situationsabhängig. Existierende Verfahren können eventuell noch optimiert werden, um noch effizienter auf spezielle Situationen zu reagieren. Diese Schlussfolgerung entsteht, wenn man wichtige Kriterien für die Effizienz von Algorithmen heranzieht. Anschließend werden sie mit bestehenden Sortieralgorithmen verglichen.
Zunächst wird das Sortierproblem und die gängigen Vergleichskriterien definiert. Daraus lässt sich die Qualität der bereits existierenden Algorithmen einordnen. Die Vergleichskriterien sind spezifische Grundsätze von Algorithmen wie die Zeitkomplexitäten der Laufzeit oder auch der Verbrauch von Speicher in einem System. Letzteres ist vor allem für Computer mit geringem Hauptspeicher essentiell. Sortierverfahren werden als stabil oder instabil sowie vergleichsbasiert beziehungsweise nicht vergleichsbasiert eingestuft. Diese Kategorisierung erklärt bestimmte Verhaltensmuster von Algorithmen.
Außerdem ist es wichtig, das Teile-und-Beherrsche-Prinzip zu verstehen, da es bei vielen Sortieralgorithmen ein essentielles Kriterium für die Lösbarkeit und die Performanz darstellt.
Die verwendeten Datenstrukturen in einem Algorithmus beeinflussen Performanz und Speicherverbrauch.

Index Terms—Sortierproblem, Algorithmus, Bubblesort, Mergesort, Radixsort, Effizientes Sortieren.

1 Einleitung

1.1 Motivation

Mehr als ein Viertel der kommerziell aufgewendeten Rechenzeit wird für das Sortieren von Daten verbraucht. Dies zeigen Untersuchungen von Computerherstellern und Computernutzern seit vielen Jahren. Aus diesem Grund wurden bereits große Anstrengungen unternommen, um das Sortieren von Daten zu optimieren. Noch immer wird stetig versucht, die Verfahren weiter zu verbessern. [5, vgl. S. 79]
Viele Informatiker betrachten das Sortieren von Daten sogar als das elementarste Problem im Bereich der Algorithmen. Diese Denkweise entsteht aus dem Umstand, dass viele Algorithmen neben ihrem eigentlichen Zweck auch ihre Daten sortieren. [2, vgl. S. 146]
Beispiele für das nebenläufige Sortieren von Daten sind vor allem verbesserte Datenstrukturen. Wenn Daten beim Einfügen in eine Liste bereits sortiert werden, fällt das Suchen bestimmter Werte leichter. [7, vgl. S. 69]
In Abbildung 1 wird dieser Vorteil des Sortierens grafisch veranschaulicht.
Trotz dieses Engagements sind weiterhin wichtige Probleme beim Sortieren einer Menge von Daten ungelöst. [5, vgl. S. 79]
Aus diesem Grund distanziert sich diese Arbeit von einer bloßen Auflistung bestehender Sortierverfahren und deren Vorzüge. Vielmehr befasst sie sich konkret mit dem Sortierproblem und dessen effizienter Lösung.



Suche nach der 7.

Abbildung 1: Beispiel: Binär-Suche in einem sortierten Array. [7, vgl. S. 69]

1.2 Forschungsfrage

Diese Arbeit beschäftigt sich mit der Forschungsfrage, inwiefern eine optimale Lösung für das Sortier-

problem gefunden werden kann.

1.3 Vorgehen / Methode

Zuerst wird das Sortierproblem definiert, um das eigentliche Problem zu veranschaulichen. Hierbei werden auch einige Grundlagen der Algorithmik erklärt. Daraufhin werden einige gängige Sortierverfahren dargestellt und verglichen. Dies dient dem Vergleich bestehender Lösungen mit dem existierenden Problem. Auf dieser Basis wird anschließend erarbeitet, wie effizientes Sortieren abläuft. Im letzten Kapitel werden die gewonnenen Ergebnisse noch einmal zusammengefasst dargestellt. Abschließend wird daraus ein Fazit gezogen.

2 Das Sortierproblem

Dieses Kapitel definiert das eigentliche Sortierproblem, das jeder Sortieralgorithmus löst. Hierbei werden Randbedingungen abgesteckt und das Ziel vorgegeben. Die Schwierigkeit, ein geeignetes Verfahren zu finden, ist nicht im Ergebnis begründet, sondern im Lösungsweg.
Das Sortierproblem definiert die Gegebenheiten und das Ziel eines jeden Sortieralgorithmus. Die Gegebenheiten entsprechen einer Menge von Sätzen, oder im Englischen auch Items genannt. Jeder dieser Sätze besitzt eine Schlüsselkomponente, nach der sortiert wird. Schlüssel lassen sich in bestimmter Weise sinnvoll anordnen durch eine sogenannte Ordnungsrelation. Diese kann bei Zahlen beispielsweise „$<$“ oder „$\leq$“ entsprechen. Bei Schlüsseln im Zeichenkettenformat wird meist alphabetisch sortiert. [5, vgl. S. 79]
Die Sätze können auch weitere Komponenten besitzen. Sie werden beim Sortieren allerdings nicht als primärer Sortierschlüssel verwendet, oder sind für den Sortiervorgang gänzlich irrelevant. [5, vgl. S. 79]
Ein Beispiel zur Verdeutlichung ist das Sortieren eines Arrays von Klassen in der objektorientierten Programmierung. Gegeben sei eine Klasse Person mit folgenden Eigenschaften:

- Vorname
- Nachname
- Alter
- Postleitzahl

Nun stellt die Klasse Person einen Datensatz dar. Soll nun ein Array mit Personen sortiert werden, müssen die einzelnen Datensätze miteinander verglichen werden. Würde man primär nach dem Alter sortieren, wäre das Alter der Schlüssel und die restlichen Daten wären nur Komponenten, die für das Sortieren irrelevant sind.
Mathematisch entspräche dies einer Folge von Sätzen $s_1 \dots s_N$. Jeder dieser Sätze s_i besitzt einen Schlüssel k_i. Nun gilt es, eine Permutation π der Sätze 1 bis n zu finden, sodass die Schlüssel in gewünschter Reihenfolge angeordnet sind. Bei ganzen Zahlen wäre dies beispielsweise entweder aufsteigend oder absteigend. Daraus ergibt sich eine mathematische Formulierung wie in Gleichung 1 gezeigt. [5, vgl. S. 79]

$$k_{\pi(1)} \leq k_{\pi(2)} \leq \dots \leq k_{\pi(n)} \tag{1}$$

[5, vgl. S. 79]

Diese Definition lässt für einige reale Szenarien viele Details und damit Fragen offen. Es bleibt beispielsweise unbeantwortet, wie groß die Anzahl der Elemente n ist. Weiterhin geht nicht hervor, ob diese Anzahl an Elementen noch in den Hauptspeicher passen. [5, vgl. S. 79f]
Jedoch soll sich die Seminararbeit nicht primär mit Definitionen möglicher Szenarien auseinandersetzen. Vielmehr geht es um das eigentliche Problem des Sortierens. Aus diesem Grund wird hier von ganzzahligen Schlüsseln oder Buchstaben ausgegangen, die sortiert werden sollen.

3 Grundlagen der Algorithmik

In diesem Kapitel werden die wichtigsten Grundlagen der Algorithmik erklärt.

3.1 Zeitkomplexitäten von Algorithmen

Die Laufzeiten von Algorithmen lassen sich in Best-Case, Worst-Case und Average-Case untergliedern. Dem Namen entsprechend steht der Best-Case für die beste Zeitkomplexität, in der ein Algorithmus terminiert. Worst-Case und Average-Case funktionieren nach demselben Vorgehen und definieren den schlechtesten beziehungsweise durchschnittlichen Fall. Die Funktion dieser Laufzeit $T : \mathbb{N} \to \mathbb{R}^+$, ist abhängig von der Problemgröße $n \in \mathbb{N}$. [7, vgl. Kapitel 2.3]
Wenn beispielsweise eine Liste von Elementen sortiert werden soll, wäre n die Länge der Liste. Die Länge einer Liste ist wiederum definiert durch die Anzahl der enthaltenen Elemente.
Bei soeben genannten Zeitkomplexitäten ist vor allem das Wachstum der Funktionen mit zunehmender Problemgröße n von Bedeutung. Hierzu wurden die sogenannten Landau-Symbole eingeführt.

Diese Funktionsklassen stellen das asymptotische Wachstumsverhalten der Funktionen dar. Unterschieden wird zwischen Groß-O (O), Omega (Ω) und Theta (Θ) Notation. Groß-O beschreibt die obere Grenze der Funktion und Omega die untere. Das Resultat aus beiden wird durch Theta dargestellt. [7, vgl. S. Kapitel 2.3]
Diese Arbeit betrachtet lediglich das Worst-Case-Szenario, da weitere Betrachtungen hier zu weit führen würden. Des Weiteren wird aus denselben Gründen nur mit der Groß-O Notation gearbeitet, um die obere Grenze aufzuzeigen.
Die Darstellung der Notationen erfolgt analog einem mathematischen Funktionsaufruf und wird beispielsweise durch $O(n)$ dargestellt. Folgendes Beispiel (Algorithmus 1) zeigt eine Python Funktion, bei der sich die Laufzeit ihres Algorithmus durch n nach oben hin beschränken lässt, wobei n die Problemgröße ist.

```
n = 100

def GrossOVonN():
  for index in range(n):
    print(index)
```

Listing 1: Python Beispiel $O(n)$

Bei der Funktion GrossOVonN() wird über das Zahlenintervall $[0 : n[$ iteriert, wobei n in diesem Beispiel 100 ist. Somit entspricht die Problemgröße der Größe des Zahlenintervalls. Daraus entsteht eine Komplexität von $O(n)$, da das Zahlenintervall n Zahlen beinhaltet.

3.2 Speicherbedarf

Es existieren Sortieralgorithmen, die keinen zusätzlichen Speicherplatz benötigen und deren Speicherbedarf konstant ist. Dieses Sortieren wird in-place genannt. Gegenteilig kann auch out-of-place sortiert werden. Hierbei wird ein Speichermehrbedarf benötigt, der von der Eingabegröße abhängt. Für Systeme, die viel Hauptspeicher zur Verfügung haben, ist der Verbrauch von zusätzlichem Speicher meist nicht relevant. Allerdings ist das Sortierproblem für alle Arten von Computer zu betrachten, so auch für Embedded Devices oder Computern mit älterer Hardware. Diese besitzen meist nur ein sehr begrenztes Volumen an Arbeitsspeicher. Aus diesem Grund muss jedes System individuell betrachtet werden. Dadurch leitet sich ab, wie viel zusätzlicher Speicher reserviert werden kann.
Typische Vertreter von in-place Verfahren sind Bubblesort (siehe Kapitel 4.1) und Quicksort. [5, vgl. Kapitel 2.1 und 2.2]
Beispiele für out-of-place Sortierverfahren sind Mergesort (siehe Kapitel 4.2) und Radixsort (siehe Kapitel 4.3). [5, vgl. Kapitel 2.4 und 2.5]

3.3 Stabiles und nicht stabiles Sortieren

Eine Sortierung ist stabil, wenn die Reihenfolge der Elemente mit identischen Schlüsseln gleich bleibt, vergleiche hierzu Gleichung 2. Hier ist wie üblich die Problemgröße als n definiert. Des Weiteren stellen i und $i+1$ benachbarte Elemente in einer fertig sortierten Liste dar. Die Größe a repräsentiert wiederum jeweils ein Schlüsselelement. Pi (π) zeigt wie bereits in Kapitel 2 die Permutationsfunktion, welche den Index eines Elements in der ursprünglich unsortierten Liste angibt. Dieser Index ist wider Erwarten in diesem Beispiel eins basiert. Somit besitzt das erste Element der Liste den Index eins.

$$\forall 1 \leq i < n : a_{\pi(i)} = a_{\pi(i+1)} \Rightarrow \pi(i) < \pi(i+1) \quad (2)$$

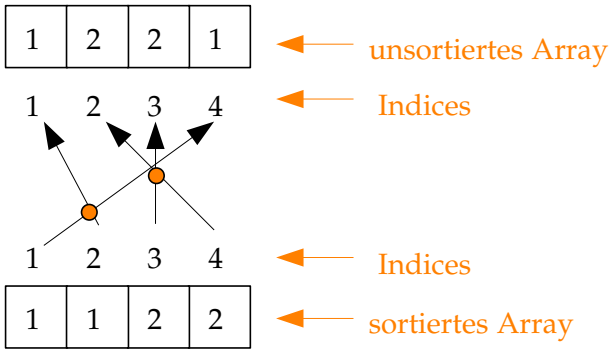
Im Folgenden wird die Bedeutung anhand eines Beispiels deutlich gemacht. Es sei ein Array mit der Schlüsselfolge 1, 2, 2, 1 gegeben. Nach der Sortierung haben die Schlüssel die Reihenfolge 1, 1, 2, 2. Hierbei ist interessant, welches der ursprüngliche Index der nun sortierten Elemente ist. Dieser ist hier ebenfalls eins-basiert. Beispielsweise ist ein Sortieralgorithmus nicht stabil, wenn die alten Indices der sortierten Liste analog Gleichung 3 aufgebaut sind.

$$\pi(1) = 4 \quad \pi(2) = 1 \quad \pi(3) = 3 \quad \pi(4) = 2 \quad (3)$$

Die Stabilitätsbedingung wird zwei Mal verletzt, da die eins die vorher an Index vier war, nun auf Index eins liegt. Damit hat sie die andere eins, die vorher an Stelle eins war, nach hinten verschoben auf Index zwei. Dies stellt eine Verletzung der Bedingung dar, denn beide Schlüssel besitzen den gleichen Wert. Ebenso verhält es sich mit den beiden Zweien. Zur Verdeutlichung dessen dient Abbildung 2.
Hingegen ist ein Algorithmus stabil, wenn die neue Reihenfolge nun so definiert ist wie in Gleichung 4

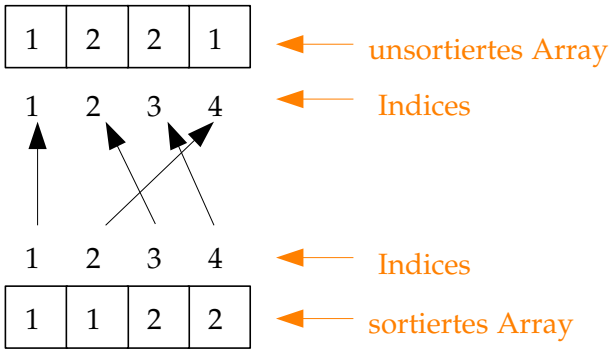
$$\pi(1) = 1 \quad \pi(2) = 4 \quad \pi(3) = 2 \quad \pi(4) = 3 \quad (4)$$

Hier wird kein gleichwertiger Schlüssel verdrängt (siehe Abbildung 3). Somit handelt es sich um einen stabilen Algorithmus. [7, vgl. S. 6]



Die Stabilitätsbedingung wird
zweimal verletzt

Abbildung 2: Beispiel für instabiles Sortieren. [7, vgl. S. 6]



Stabiles Sortieren der Zahlenfolge

Abbildung 3: Beispiel für stabiles Sortieren. [7, vgl. S. 6]

3.4 Vergleichsbasierte und nicht vergleichsbasierte Sortierverfahren

Sortieralgorithmen arbeiten entweder vergleichsbasiert oder nicht vergleichsbasiert. Vergleichsbasierte Algorithmen werden im englischen *comparison sorts* genannt. Sie heißen so, weil ihr Vorgehen zum Sortieren auf Vergleichen der Schlüsselelemente beruht. Es ist mathematisch erwiesen, dass sie im Worst-Case keine bessere nach oben beschränkte Laufzeitkomplexität haben können als $O(n * log_2(n))$.

Es existieren allerdings Algorithmen, für die diese Grenze von $O(n * log_2(n))$ nicht gilt. Hierbei wird in linearer Zeit sortiert. Deren Vorgehen basiert allerdings nicht auf Vergleichen. Somit werden diese nicht vergleichsbasierte Algorithmen genannt. [2, vgl. Kapitel 8]

3.5 Teile und Beherrsche

Das Prinzip des Teilens und Beherrschens (engl. *divide and conquer*) ist ein allgemeingültiger Ansatz für das Lösen eines Problems per Algorithmus. Ein Problem wird dabei in kleinere gleichartige Probleme geteilt. Diese werden dann rekursiv gelöst, bis die betrachteten Probleme so klein sind, dass sie sofort trivial gelöst werden können. Dabei besteht die Problemlösung aus drei Teilen:

- **Teile**: Ein Problem wird in kleinere Probleme zerlegt.
- **Beherrsche**: Die Teilprobleme werden rekursiv gelöst.
- **Verbinde**: Die Einzellösungen der Teilprobleme werden anschließend zu einem Ergebnis des Gesamtproblems zusammengefügt.

Bei den Sortieralgorithmen sind hierfür die bekanntesten Beispiele der Mergesort (siehe Kapitel 4.2) und der Quicksort. [7, vgl. Kapitel 7]

3.6 Datenstrukturen

Die Wahl der richtigen Datenstruktur ist sehr wichtig für jeden Algorithmus. Hier wird die Speicherstruktur festgelegt, die wiederum für die Zeitkomplexitäten der jeweiligen Datenstruktur da sind. Es existieren verschiedene Zeitkomplexitäten für unterschiedliche Operationen. Datenstrukturen halten Daten. Somit müssen diese Daten eingefügt, gelesen und gelöscht werden. Außerdem kann über sie iteriert werden. Für die Zeitkomplexitäten wird wieder auf die Groß-O Notation zurückgegriffen (siehe Kapitel 3.1). Beispielsweise benötigt ein Test, ob ein Element in einer Liste existiert, $O(n)$ Schritte, während es bei einer Hash-basierten Datenstruktur $O(1)$ Schritte bedarf.

Aus diesem Grund muss bei jedem Algorithmus abgeschätzt werden, welche Operationen vorherrschend sind. Wenn primär iteriert wird, setzt man am Besten auf eine Liste. Wenn Daten existentiell abgefragt werden, empfiehlt sich eher eine Hash-basierte Datenstruktur. [3, vgl. S. 50]

In höheren Programmiersprachen entstehen durch das Sortieren oftmals verbesserte Datenstrukturen neben den grundlegenden. Das Sortieren wird als Subroutine implementiert und daraus beispielsweise eine sich selbst sortierende Liste gebaut. Diese sortiert das Element nach dem Einfügen an der richtigen Stelle ein. [3, vgl. S. 76ff]

4 Bestehende Sortieralgorithmen

Im folgenden Kapitel sind einige gängige Sortierverfahren aufgezeigt und beispielhaft implementiert. Zu beachten ist, dass dies nicht alle existierenden Sortierverfahren sind. Es existieren zu viele für diese Arbeit, wobei jedes meist noch mehrere Unterarten besitzt, die in bestimmten Szenarien für mehr Effizienz sorgen.
Die folgenden Verfahren sind bewusst ausgewählt, um ein möglichst breites Spektrum der Grundlagen aus Kapitel 3 abzudecken und dadurch weiteres Grundverständnis aufzubauen.

4.1 Bubblesort

Der Bubblesort ist ein sehr einfacher Sortieralgorithmus. Er benötigt keinen zusätzlichen Speicherplatz, findet also in-place statt. Zusätzlich ist die Sortierung stabil.
Beim Bubblesort Algorithmus wird nur das Vertauschen von benachbarten Datensätzen erlaubt. Die folgenden Erklärungen und Beispiele behandeln immer den Fall einer aufsteigenden Sortierung. Wenn beispielsweise $a[i].key > a[i+1].key$, so sind diese beiden Datensätze in absteigender Reihenfolge angeordnet. Folglich müssen sie vertauscht werden, um die aufsteigende Permutation der beiden Elemente zu erreichen. Die Vertauschungen passieren innerhalb von zwei verschachtelten Schleifen, die beide über die Liste iterieren.
Die äußere Schleife durchläuft die Liste und nach jedem Listendurchlauf werden die Elemente der Größe nach absteigend an das Ende verschoben. Somit sind immer die hintersten Stellen zuerst sortiert. Daraus folgt, dass nach jedem Durchlauf der äußeren Schleife ein Element seinen vorgesehenen Platz erreicht hat.
Die innere Schleife iteriert über die Liste und stellt somit einer darunter geschachtelten if-Abfrage die richtigen Indices beziehungsweise den aktuellen Datensatz der Liste bereit. Zu beachten ist, dass nicht über die gesamte Liste iteriert werden muss, da die letzten Elemente bereits sortiert wurden. Hierbei kann die Anzahl der Elemente daraus ermittelt werden, wie oft die äußere Schleife bereits durchlaufen wurde. Nun wird in der if-Abfrage geprüft, ob eine Vertauschung erfolgen soll. Falls dies der Fall ist, wird diese vorgenommen. Diese Prozesse werden so lange wiederholt, bis die äußere Schleife mit ihrer Iteration über die Liste fertig ist. Zur beispielhaften Veranschaulichung dient Abbildung 4. Hier wird eine Liste mit Hilfe des Bubblesort sortiert, wobei die farbigen Pfeile Vertauschungen von Zahlen angeben.
Der folgende Python Code (Algorithmus 2) veranschaulicht die Funktionsweise des Bubblesort.

```
def bubblesort(ord_list):
  len_list = len(ord_list)

  for i in range(len_list-1):
    for j in range(len_list-i-1):
      if ord_list[j] > ord_list[j+1]:
      # vertauschen:
      ord_list[j], ord_list[j+1] = ord_list[j
    +1], ord_list[j]
```

Listing 2: Bubblesort Algorithmus in Python

Bei einem Bubblesort Algorithmus kann man auf eine quadratische Komplexität schließen. Aus diesem Grund ergibt sich für die Komplexität der Wert $O(n^2)$. Aufgrund der schlechten Zeitkomplexitäten ist der Bubblesort kein gutes elementares Sortierverfahren. [5, vgl. Kapitel 2.1.4]

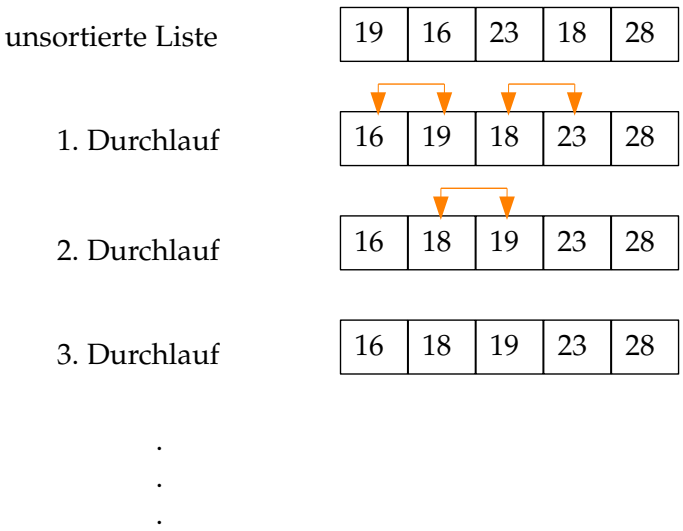


Abbildung 4: Bubblesort Beispiel

4.2 Mergesort

Der Mergesort ist ein stabiles Sortierverfahren mit einer Komplexität von $O(n * log_2(n))$. Er ist ein vergleichsbasierter Sortieralgorithmus. Für die Sortierung wird das Teile-und-Beherrsche-Prinzip (siehe Kapitel 3.5) angewendet. Dabei wird ein zu sortierendes Array in zwei gleich große Hälften zerteilt. Anschließend werden diese dann rekursiv weiter zerteilt, bis nur noch Felder der Länge eins übrig bleiben. Diese Teilstücke werden dann sortiert und wieder zu einem Feld zusammengesetzt. Durch das Zerteilen, Umkopieren und Zusammenfügen ist

auch selbsterklärend, dass ein Mergesort nicht in-place stattfinden kann. Es wird zusätzlicher Speicherplatz benötigt. [7, vgl. Kapitel 7.1] Der Mergesort Algorithmus wird in Abbildung 5 veranschaulicht. Die farbigen Pfeile kennzeichnen hierbei das Aufteilen und Verschmelzen der einzelnen Listen.

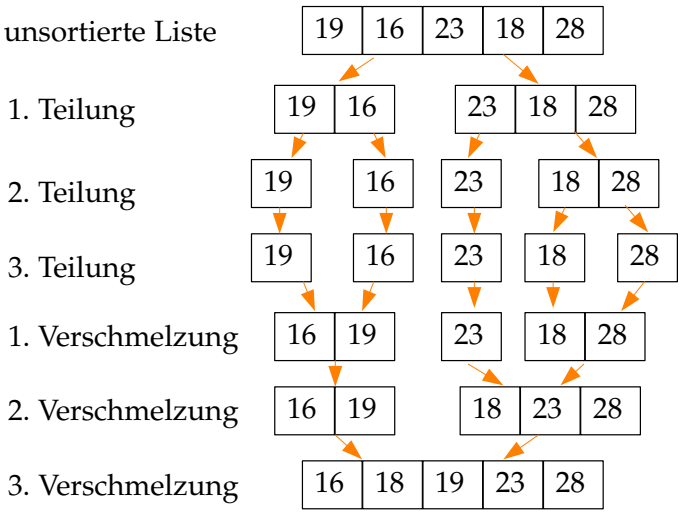


Abbildung 5: Mergesort Beispiel, basierend auf [4]

Im folgenden Python-Code (Algorithmus 3) wird die Funktionsweise des Mergesort aufgezeigt.

```
def mergesort(list_to_sort):
  ########################
  # 1. Teil
  ########################
  if len(list_to_sort) > 1:
    # Die Mitte der Liste berechnen
    mid = len(list_to_sort)//2
    # Die linke Haelfte der Liste berechnen
    left_list = list_to_sort[:mid]
    # Die rechte Haelfte der Liste berechnen
    right_list = list_to_sort[mid:]
    # Rekursiv teilen bis nur noch ein Element
     vorhanden
    left_list = mergesort(left_list)
    right_list = mergesort(right_list)

  ########################
  # 2. Teil
  ########################
  list_to_sort = []

  while len(left_list) > 0 and len(right_list)
     > 0:
    if left_list[0] < right_list[0]:
      list_to_sort.append(left_list[0])
      left_list.pop(0)
    else:
      list_to_sort.append(right_list[0])
      right_list.pop(0)

  for i in left_list:
    list_to_sort.append(i)
  for i in right_list:
    list_to_sort.append(i)

  return list_to_sort
```

Listing 3: Mergesort Algorithmus in Python

Der Algorithmus 3 umfasst zwei Teile. Der erste Teil ist für das rekursive Teilen verantwortlich, während Teil zwei das Sortieren und Zusammenfügen übernimmt. Somit sind alle drei Teile zur Problemlösung anhand des Teile-und-Beherrsche-Prinzips abgedeckt und das Sortierproblem kann als gelöst betrachtet werden.

4.3 Radixsort

Der Radixsort ist ein nicht vergleichsbasiertes Sortierverfahren. Es existieren mehrere Varianten. Hier wird speziell der Bucketsort, auch Binsort genannt, aufgegriffen. Dieser ist ein stabiles Sortierverfahren, funktioniert aber nicht in-place. Generell ist das Prinzip des Radixsort, das Sortierproblem der Komplexität n zu lösen. Allerdings gelten hier Einschränkungen bei den Schlüsselelementen. Um den Radixsort effizient anwenden zu können, sollten die Schlüsselelemente die gleiche Länge l besitzen. Es ist auch möglich, unterschiedliche Schlüssellängen zu verwenden, allerdings wird dadurch eine zusätzliche Abfrage benötigt. Dies macht den Algorithmus zwar lauffähig, verschlechtert allerdings die Laufzeitkomplexität. Zusätzlich müssen sie einem endlichen Alphabet angehören. Dieses Alphabet hat die Länge m.

- m = 10: Wenn die Schlüssel Dezimalzahlen sind.
- m = 2: Wenn die Schlüssel Dualzahlen sind.
- m = 26: Wenn die Schlüssel Wörter aus dem Alphabet sind.

Wenn die Schlüssel, wie oben bereits angedeutet, Dezimalzahlen sind, dann ist $m = 10$. Dies beinhaltet das Alphabet aller Dezimalzahlen von 0 bis 9. Analog verhält es sich bei Dualzahlen, da hier nur zwei Zahlen im Alphabet existieren. Im deutschen Alphabet befinden sich 26 Buchstaben von A bis Z. Somit ist $m = 26$.
Daraus ergibt sich erstmals eine lineare Komplexität von $O(l * (m + n))$. Diese gute Komplexität ist nur möglich, weil nicht mit Schlüsselvergleichen gearbeitet wird.
Eine zusätzliche Schwäche dieser Radixsort Variante ist der vermehrte Speicherverbrauch. Durch die vielen Einschränkungen lohnt sich der Radixsort

prinzipiell nur in Spezialfällen, wenn die Schlüsselelemente die richtige Form haben. [5, vgl. Kapitel 2.5]
Im Listing (4) ist zur Veranschaulichung der Radixsort in Python programmiert dargestellt. Dieser Algorithmus kann alle Schlüssellängen interpretieren, vorausgesetzt es handelt sich um Dezimalzahlen.

```
m = 10

def radixSort(list_to_sort):
  maximum_number = max(list_to_sort)
  l = len(str(maximum_number))

  for i in range(1, l):
    temp_list = [[] for elem in range(m)]
    for number in list_to_sort:
      if i+1 <= len(str(number)):
        digit = int(str(number)[-i-1:-i])
      else:
        digit = 0
      temp_list[digit].append(number)

    list_to_sort = []
    for digit_number_sort_list in temp_list:
      list_to_sort = list_to_sort +
    digit_number_sort_list

  return list_to_sort
```

Listing 4: Radixsort Algorithmus in Python

Betrachtet man den Radixsort Algorithmus in Listing (4), so wird zuerst die Größe m definiert. Diese ist, wie am Anfang dieses Kapitels erklärt, die Länge des Alphabets. Das Alphabet ist mit 10 definiert, da in diesem Beispiel Dezimalzahlen als Schlüsselelemente verwendet wurden. Der radixSort-Funktion wird nun die zu sortierende Liste übergeben.
Bei einem Radixsort werden die Zahlen, die die Schlüsselelemente darstellen, aufgeteilt in Einzelzahlen. Danach werden diese in Kategorien eingeteilt, welche wiederum nach einigen Iterationen die sortierte Liste ergeben. Die Anzahl der Iterationen entspricht der Stellenanzahl (l) der größten Zahl in der Liste. Wenn eine Liste nun aus den Zahlen 1, 3, 100 und 205 bestehen würde, wäre $l = 3$, da 205 die größte Zahl ist und aus drei einzelnen Zahlen besteht. Nun wird über die Liste iteriert und von jeder Zahl wird die erste Einzelzahl extrahiert. Wie bereits erwähnt, handelt es sich um die niederwertigste. Anschließend wird eine Liste mit m leeren Listen erzeugt. Diese repräsentieren die einzelnen Behältnisse von 0-9. Der Bucketsort ist nach diesen Behältnissen (engl. buckets) benannt. Die Zahlen werden im Ganzen in den entsprechenden Listen jeweils hinten angehängt. Somit bleibt die Stabilität des Sortierverfahrens bestehen. Die Auswahl der Liste geschieht nach dem Wert der extrahierten Einzelzahlen. Wenn dies beendet ist, sollte jede Zahl in einer der zuvor leeren Listen vorkommen. Als letzter Schritt der Iteration werden die einzelnen Listen zu einer großen verschmolzen. Dies stellt die neue Liste dar.
Jetzt folgt die zweite Iteration mit der nächst niederwertigsten Zahl innerhalb der Schlüsselzahlen. Dies wird wiederholt, bis von jeder Zahl die höchstwertigste Einzelzahl klassifiziert wurde. Nun ist die Liste sortiert.
Dieser Prozess ist in Abbildung 6 noch einmal beispielhaft grafisch dargestellt. Die farbig markierten Einzelzahlen bilden die Stellen der Gesamtzahl ab, die in der aktuellen Iteration betrachtet werden.

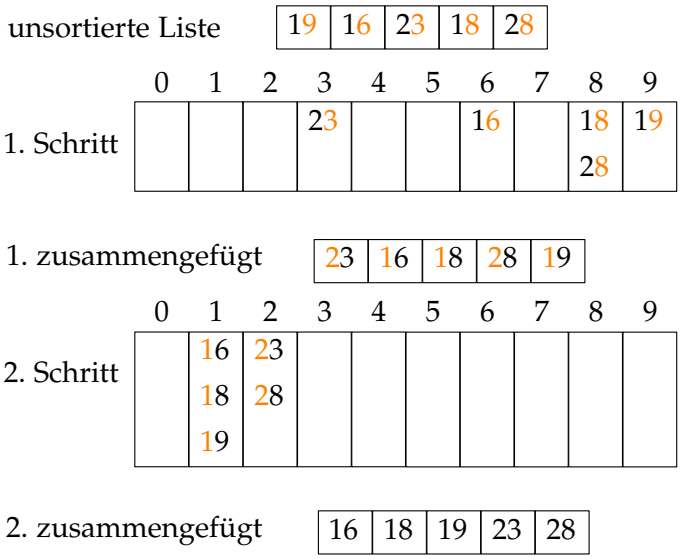


Abbildung 6: Radixsort Beispiel, basierend auf [6]

4.4 Sortierverfahren im Überblick

Im folgenden Absatz werden die bisherigen Sortieralgorithmen noch einmal gegenübergestellt (siehe hierzu Tabelle 1). Bei den Platzhaltern der Laufzeit steht n für die Problemgröße, l für die Anzahl der möglichen Zeichen (Basis) und m für die Anzahl an möglichen Schlüsseln.

Name	Laufzeit	stabil	in-place
Bubblesort	$O(n^2)$	ja	ja
Mergesort	$O(n * log_2(n))$	ja	nein
Radixsort	$O(l * (m + n))$	ja	nein

Tabelle 1: Überblick der behandelten Sortieralgorithmen

5 Effizientes Sortieren

Bei der Findung eines effizienten Sortieralgorithmus stellt sich zunächst die Frage, was unter einem effizienten Algorithmus zu verstehen ist. Algorithmen zählen nicht nur als effizient, wenn sie schnell mit dem gewünschten Ergebnis terminieren, sondern auch wenn sie Speicherplatz sparen. Die Eigenschaft der Effizienz steht bei Algorithmen in einer starken Wechselbeziehung zur benutzten Datenstruktur, da diese für die Speicherverwaltung und gleichzeitig auch für die Zugriffszeiten verantwortlich ist. [7, vgl. S. 36]

Wenn man nun die Vor- und Nachteile der Algorithmen miteinander vergleicht, wird schnell klar, dass keiner dieser Algorithmen als der Beste angesehen werden kann. Vorteile in einer Kategorie, beispielsweise der Zeitkomplexität, gehen mit Nachteilen einer anderen Kategorie einher. Beim Radixsort (Kapitel 4.3) bedeutet dies beispielsweise signifikante Einschränkungen und Speichermehrverbrauch. Ein zweiter Algorithmus, der hier nicht ausformuliert wurde, der in linearer Zeit terminiert, ist der Countingsort. Bei diesem wird ebenfalls die Zeitkomplexität mit mehr Speicher ausgeglichen. Ebenso beim Mergesort (Kapitel 4.2), bei dem die zu sortierenden Listen immer wieder aufgeteilt werden.

In-Place Sortierverfahren, wie der Bubblesort, haben ihre Schwachstellen oft in der Laufzeitkomplexität. Hier wird kein zusätzlicher Speicher verbraucht, die Ausführung dauert allerdings länger.

Des Weiteren ist es bei einigen Algorithmen von Bedeutung, inwieweit die Liste bereits vorsortiert ist. Eine bereits komplett vorsortierte Liste kann als Spezialfall angenommen werden. Hier kommen oft die Best-Cases (vergleiche Kapitel 3.1) der Algorithmen zur Anwendung.

Hinzu kommt, dass schon viele Arbeiten, die die Vor- und Nachteile einzelner Sortieralgorithmen in bestimmten Situationen miteinander vergleichen, existieren. Je nach Situation ergeben sich unterschiedliche Ergebnisse der Effizienz. Zum Beispiel ist laut Cook und Kim bei kleinen und nahezu bereits vorsortierten Listen der Straight Insertion Sort der beste Algorithmus. [1, vgl. Kapitel 5]

Somit ist anhand der Grundlagenvergleiche und der Ansicht gängiger Sortieralgorithmen klar erkennbar, dass für das Sortierproblem noch keine hinreichend effiziente Lösung entwickelt wurde, die für alle Gegebenheiten optimal ist. Es werden immer wieder neue Algorithmen entwickelt, um spezielle Anwendungsfälle besser zu lösen. Es existiert jedoch kein Optimum, das sich für alle eintretenden Szenarien eignet.

6 Conclusion

Diese Arbeit hat gezeigt, dass es keinen idealen Sortieralgorithmus für jede Situation gibt. Somit ist es situationsabhängig, welcher Algorithmus angewendet werden sollte. Beispielsweise kann bei Servern in Rechenzentren mit sehr großzügigem Arbeitsspeichervolumen der Fokus auf einen höheren Speicherverbrauch zu Gunsten schnellerer Zugriffszeiten gelenkt werden. Demnach könnte diese Randbedingung hier wenig bis keine Rolle spielen.
Im Gegensatz dazu ist die Speicherbelegung bei Embedded Geräten von zentraler Bedeutung, da nur begrenzt Speicher zur Verfügung steht.
Ebenfalls kritisch zu betrachten ist die Wahl der Datenstruktur, die für einen Algorithmus verwendet wird. Hier ergeben sich Unterschiede bei den Zugriffszeiten, den Zeiten für eine Iteration und für das Schreiben von Daten. Um auch die richtige Version eines Algorithmus zu implementieren, empfiehlt es sich daher, die Wahl der richtigen Datenstruktur zu beachten.
Schlussendlich gibt es für jeden Anwendungsfall einen gut geeigneten Algorithmus zum Sortieren. In vielen Fällen ist dieser bereits vorhanden. Falls nicht, hat man die Möglichkeit, bei bereits bestehenden Algorithmen die Datenstrukturen auf die eigenen Wünsche anzupassen, um so mehr Performanz zu generieren. Ebenfalls ist es möglich, bestehende Algorithmen zu vermischen und daraus einen neuen Algorithmus zu bauen.
Folglich ist das Sortierproblem zwar gelöst, dennoch ist nicht erwiesen, ob es nicht noch bessere Verfahren in Zukunft geben wird.

LITERATUR

[1] Curtis R. Cook und Do Jin Kim. „Best Sorting Algorithm for nearly Sorted Lists“. In: (1980).
[2] Thomas H. Cormen u. a. *Algorithms*. Third Edition. 2009.
[3] Rick van Hattem. *Mastering Python*. Packt Publishing, 2016.
[4] *Mergesort*. URL: https://studyflix.de/informatik/mergesort-1324.
[5] Thomas Ottman und Peter Widmayer. *Algorithmen und Datenstrukturen*. 6. Aufl. Springer Verlag, 2017.
[6] *Radix Sort*. URL: https://studyflix.de/informatik/radix-sort-1408.
[7] Karsten Weicker und Nicole Weicker. *Algorithmen und Datenstrukturen*. Springer Verlag, 2013.