Bericht zum Modulabschluss

Mergesort und Combsort

T. Hadamczik und T. Höfting

16. Januar 2020

Inhaltsverzeichnis

1 Schnittstellendokumentation

Im Folgenden werden die Funktionen beschrieben, die in abschlussprojekt.py implementiert sind. Damit die Prozesszeit untersucht werden kann, wird das Modul `time` importiert.

1.1 `input_processing(file_location)`

Input:
`file_location` *(str)* Pfad zu der Datei, welche gelesen werden soll
Returns:
`word_list` *(list[str])* Liste mit allen durch Leerzeichen (inkl. white-space, newline, tabstop) getrennten Wörtern aus der Datei

In dieser Funktion wird eine Textdatei, die unter $< file\_location >$ gespeichert ist so verarbeitet, dass am Ende eine Liste aus allen Wörtern zurückgegeben wird, wobei kein Wort Leerzeichen, Zeilenumbrüche oder Tabstops enthält. An diesen drei Zeichen werden mithilfe der Funktion `split()` die Wörter getrennt. Dabei können leere Wörter entstehen, die anschließend noch aus der Liste entfernt werden.

1.2 `mergesort(word_list)`

Input:
`word_list` *(list[str])* Liste der zu sortierenden Wörter
Returns:
`sorted_list` *(list[str])* Liste mit den Wörtern der übergebenen Liste in richtiger Reihenfolge
`comparisons` *(int)* Anzahl der durchgeführten Vergleiche

In `mergesort(word_list)` wird eine Liste von Strings mittels des Sortieralgorithmus Mergesort sortiert, welcher im Flussdiagramm 1.1 dargestellt ist. Anhand eines Beispieldurchlaufs wird die Funktionsweise erläutert, wobei die Variable `comparisons` ignoriert wird, da sie lediglich der Zählung der Vergleiche dient. Man übergibt der Funktion eine Liste von Strings, im Beispiel die Liste

```
["Ford","Opel","Audi","VW","Opel"]
```

Um zu zeigen, dass der Algorithmus im Beispiel stabil läuft, haben die beiden Elemente "`Opel`", die den gleichen Sortierschlüssel besitzen, unterschiedliche Färbungen. Im ersten

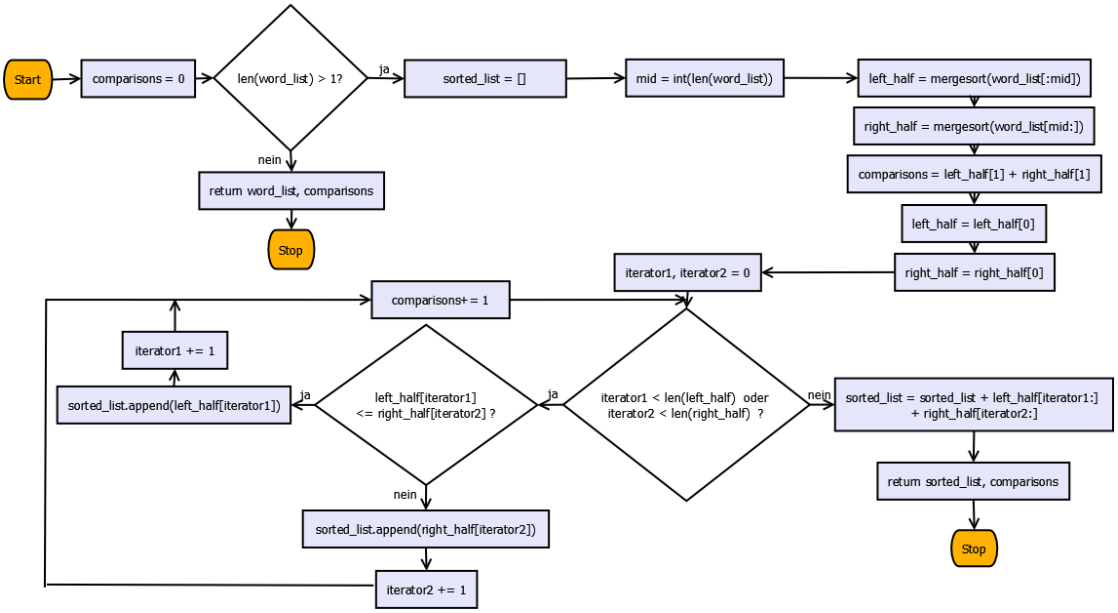

Abbildung 1.1: Flow-Chart zu `mergesort(word_list)`

Schritt wird geschaut, ob die Liste mehr als ein Element hat. Falls nicht, wird die Liste direkt zurückgegeben, da eine einelementige Liste sortiert ist. Andernfalls wird eine neue leere Liste *sorted_list* definiert, in der wir nachher die sortierten Elemente anordnen werden. Im nächsten Schritt geschieht das *Teilen*. Die Liste wird in zwei Listen

```
["Ford","Opel"], ["Audi","VW","Opel"]
```

geteilt. Im selben Schritt findet die Rekursion statt, das heißt diese beiden Listen werden jeweils mit dem Mergesort Algorithmus sortiert. Also wird

```
["Ford","Opel"]
```

aufgeteilt in

```
["Ford"],["Opel"]
```

und der Algorithmus wird darauf angewandt. Da diese zwei Listen nicht mehr als ein Element haben, ist dies die unterste Rekursionsebene und die Listen werden unverändert auf ihr zurückgegeben. Nun werden Sie ineinander sortiert. Da sie einelementig sind, muss nur ein Vergleich von "Ford“ und "Opel"gemacht werden. Nach der Interpretation von Python ist "Ford"kleiner als "Opel". Also wird die Liste

```
["Ford","Opel"]
```

zurückgegeben. Der Vorgang wird analog für die rechte Hälfte ausgeführt. Im ersten "*Teilen*“ bekommen wir die beiden Listen

```
["Audi"],["VW","Opel"]
```

`mergesort(word_list)` wird wieder auf die beiden Hälften angewandt. Dabei wird die linke Liste unverändert zurückgegeben. Die rechte Liste wiederum wird in die beiden Listen

```
["VW"],["Opel"]
```

geteilt, auf die wiederum `mergesort(word_list)` angewendet wird. Da diese nur noch einelementig sind, werden sie wieder unverändert zurückgegeben. Nun werden die beiden Listen ineinander sortiert. Es wird wieder nur ein Vergleich vollzogen, nach Python-Logik ist "Opel" kleiner als "VW". Also wird die Liste

```
["Opel","VW"]
```

ausgegeben. Diese ausgegebene Liste wird nun auf der höheren Ebene mit der Liste `["Audi"]` verglichen. Zuerst werden die jeweils ersten Elemente verglichen, also "Audi" mit "Opel". Da "Audi" kleiner ist als "Opel", wird "Audi" in der neuen sortierten Liste gespeichert. Da nun über alle Objekte der ersten Liste iteriert wurde, wird die while-Schleife abgebrochen, und die Elemente, die aus der zweiten Liste noch übrig geblieben sind werden an die sortierte Liste angefügt. Somit ergibt sich die Ausgabe

```
["Audi","Opel","VW"]
```

Wir haben jetzt also die beiden sortierten Listen

```
["Ford","Opel"],["Audi","Opel","VW"]
```

Im letzten Schritt werden diese geordneten Listen zusammengefügt. Zuerst schaut man sich die jeweils ersten Elemente "Ford" und "Audi" an. "Ford" $>$ "Audi", deswegen wird "Audi" zur Liste *sorted_list* hinzugefügt. Als nächstes wird "Ford"' mit "Opel" verglichen. "Ford" $\leq$ "Opel", also wird "Ford" zu *sorted_list* hinzugefügt. Nun wird "Opel" mit "Opel" verglichen. Es gilt "Opel" $=$ "Opel", also wird "Opel" der sortierten Liste angefügt. Da nun die linke Hälfte abgearbeitet ist, werden die restlichen Elemente der rechten Liste einfach dran gehangen. Wir erhalten somit die vollständig sortierte Liste

```
["Audi","Ford","Opel","Opel","VW"]
```

die zurückgegeben wird. Insgesamt kommen wir auf 6 Vergleiche. Die Reihenfolge von "Opel" und "Opel" wurde eingehalten.

1.3 `combsort(word_list)`

Input:
`word_list` *(list[str])* Liste der zu sortierenden Wörter
Returns:
`sorted_list` *(list[str])* Eine Liste der Wörter aus der Eingabeliste in lexiografisch aufsteigender Reihenfolge

`sum_swaps_comps` *(int)* Die Summe von durchgeführten Vergleichen und Vertauschungen

Diese Funktion bekommt eine Liste von Wörtern als Eingabe übergeben und sortiert diese mittels des Sortierverfahrens `combsort`. Die Funktionsweise des Algorithmus wird im Flussbilddiagramm 1.2 dargestellt. *word_index* bezeichnet im Diagramm den Index des jeweiligen Wortes in der Liste. Das Hochzählen der Variablen `swaps` und

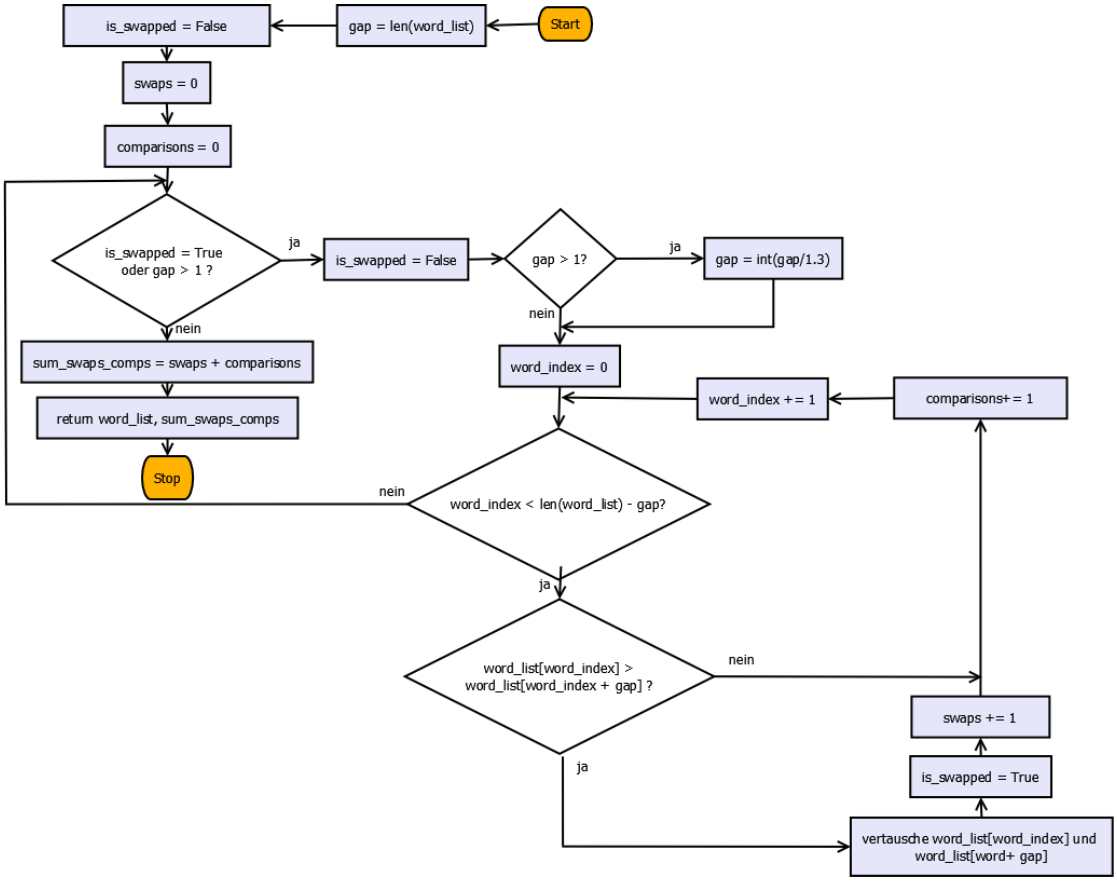

Abbildung 1.2: Flow-Chart zu `combsort(word_list)`

`comparisons` dient nur der Auswertung und wird bei der Beschreibung des Algorithmus ausgelassen. Als erstes wird der Variable `gap` die Anzahl der Listenelemente zugewiesen. Außerdem wird die Variable `is_swapped` mit `False` initialisiiert. Nun wird eine while-Schleife ausgeführt solange `is_swapped` den Wert `True` hat oder `gap` einen Wert größer als 1 hat. In einem Schleifendurchlauf wird der Wert von `is_swapped` auf `False` und der Wert von `gap` auf die nächstkleinere Ganzzahl des ursprünglichen Werts von `gap` geteilt durch 1,3 gesetzt. Dann wird innerhalb der while-Schleife eine for-Schleife durchlaufen, in der jedes Wort der Liste, für das das Wort welches [Wert von `gap`] Stellen weiter in der Liste steht exisitiert, mit diesem verglichen wird. Falls die verglichenen Wörter

noch nicht in der korrekten Reihenfolge stehen, werden sie vertauscht und der Wert von `is_swapped` wird auf `True` gesetzt. Im folgenden wird anhand eines Beispieldurchlaufes mit der unsortierten Liste

```
["VW","Ford","Opel","Opel"]
```

die Funktionsweise des Algorithmus veranschaulicht. Da die Liste vier Elemente enthält, wird `gap` auf 4 gesetzt, `is_swapped` wird zudem mit `False` initialisiert. Nun beginnt die while-Schleife und `is_swapped` wird auf `False` gesetzt sowie der Wert von `gap` (da `gap` > 1) ganzzahlig durch 1,3 geteilt und nimmt den neuen Wert 3 an. Es wird nun das erste Element der Liste ("VW") mit dem vierten Element ("Opel") verglichen. Da diese sich nicht in der richtigen Reihenfolge befinden, werden sie vertauscht und `is_swapped` auf `True` gesetzt. Man erhält die neue Liste

```
["Opel","Ford","Opel","VW"]
```

Da `gap` noch einen Wert größer 1 hat, wird `is_swapped` wieder auf `False` gesetzt und der Wert von `gap` wieder ganzzahlig durch 1,3 geteilt, nimmt also den neuen Wert 2 an. Nun wird das erste Element ("Opel") mit dem dritten Element ("Opel") der Liste verglichen. Diese befinden sich bereits in richtiger Reihenfolge, daher werden sie nicht vertauscht. Dann wird das zweite ("Ford") mit dem vierten Element ("VW")verglichen. Diese sind ebenfalls in der richtigen Reihenfolge, daher beginnt der nächste Schleifendurchlauf, da `gap` noch einen Wert größer 1 hat. `is_swapped` wird sein bestehender Wert `False` nochmals zugewiesen und `gap` nimmt nun den neuen Wert 1 an, sodass jedes Element der Liste mit seinem Nachfolger verglichen wird: Das erste Element ("Opel") wird mit dem zweiten ("Ford") verglichen und vertauscht, da die Reihenfolge nicht korrekt ist. Die Liste ist nun

```
["Ford","Opel","Opel","VW"]
```

und `is_swapped` nimmt den Wert `True` an. Dann wird das zweite Element der neuen Liste ("Opel") mit dem dritten ("Opel") verglichen. Diese befinden sich in der richtigen Reihenfolge. Als letztes wird das dritte Element ("Opel") mit dem vierten ("VW") verglichen. Diese befinden sich ebenfalls in der richtigen Reihenfolge. Zwar hat `gap` nun den Wert 1, allerdings wurde im letzten Schleifendurchlauf noch eine Vertauschung durchgeführt, sodass `is_swapped` den Wert `True` hat und ein weiterer Durchlauf mit `gap` $= 1$ durchgeführt wird. Es werden nun ("Ford") mit ("Opel"), ("Opel") mit ("Opel") sowie ("Opel") mit ("VW") verglichen. Da diese sich alle in der richtigen Reihenfolge befinden, behält `is_swapped` den Wert `False` bei und da `gap` den Wert 1 hat, wird kein weiterer Durchlauf durchgeführt und der Algorithmus terminiert.

1.4 `sort_A(word_list)`

Input:
`word_list` *(list[str])* Liste der zu sortierenden Wörter
Returns:

`comparisons` *(int)* Anzahl der ausgeführten Vergleiche
`used_time` *(float)* Ausführungszeit des Sortierens

Um `mergesort(word_list)` untersuchen zu können, brauchen wir die Funktion `sort_A(word_list)`. In ihr wird die Funktion `mergesort(word_list)` auf die übergebene Liste angewandt. Dabei wird die Zeit vor und nach dem Aufruf gemessen, und somit die Prozesszeit bestimmt. Außerdem wird die Variable `comparisons`, welche von `mergesort(word_list)` zurückgegeben wird ausgegeben um Informationen über die Anzahl der Vergleiche zu erhalten.

1.5 `sort_B(word_list)`

Input:
`word_list` *(list[str])* Liste der zu sortierenden Wörter
Returns:
`sum_swaps_comps` *(int)* Summe der ausgeführten Vergleiche und Vertauschungen
`used_time` *(float)* Ausführungszeit des Sortierens

In `sort_B(word_list)` wird der Sortieralgorithmus Combsort ausgeführt und analysiert. Dazu wird die Zeit vor und nach dem Aufruf von `combsort(word_list)`gemessen, und somit die Prozesszeit bestimmt. Außerdem wird die Variable `sum_swaps_comps`, welche von `combsort(word_list)` zurückgegeben wird und Informationen über die Anzahl der durchgeführten Vergleiche und Vertauschungen enthält, ausgegeben.

1.6 Hauptprogramm

Im Hauptprogramm ist die Interaktion mit dem Nutzer implementiert. Zuerst wird der Nutzer aufgefordert den Namen der Datei, in der sich die zu sortierenden Strings befinden in die Standardeingabe zu schreiben. Dieser wird dann `input_processing(file_location)` übergeben und zurück kommt eine Liste der zu sortierenden Wörter. Falls ein falscher Dateipfad eingegeben wurde, wird ein `FileNotFoundError` geworfen. Falls dies nicht der Fall ist werden auf die so erhaltene Liste die beiden Funktionen `sort_A(word_list)` und `sort_B(word_list)` zur Analyse der Sortierverfahren angewandt, woraufhin die Rückgabewerte der Funktionen, also die Summe von Vergleichen und Vertauschungen und die Prozesszeit geprinted werden.

2 Theoretische und experimentelle Untersuchungen

2.1 Mergesort

2.1.1 Algorithmus

Das Sortierverfahren *Mergesort* basiert auf dem Vorgang des *Mischens* (*merge*); dem Zusammenfassen von zwei geordneten Listen zu einer großen geordneten Liste. Grundlage für den Algorithmus ist das Prinzip *Teile und herrsche*. Wir zerlegen die zu ordnende Liste in zwei Teile, sortieren diese getrennt und fassen sie dann zu einer komplett sortieren Menge zusammen. Abbildung 2.1 stellt den Prozess schematisch dar. Der Algorithmus

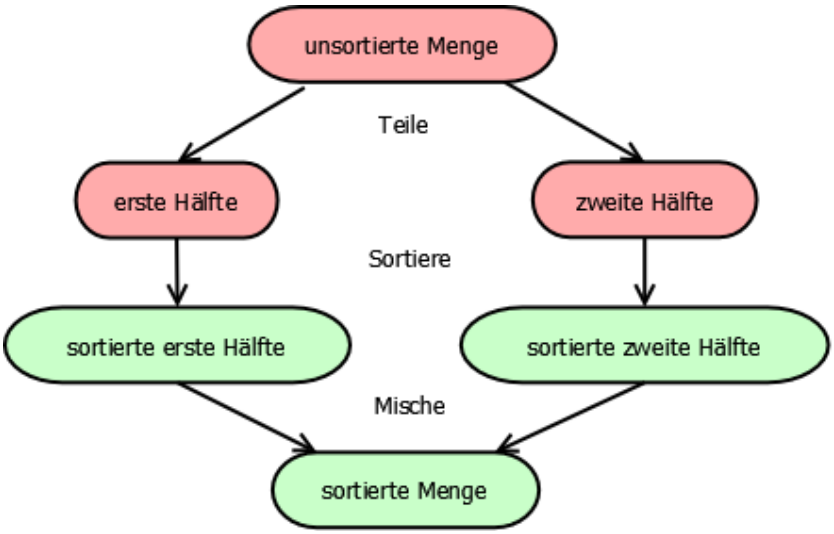

Abbildung 2.1: Schema von mergesort

wird *rekursiv* implementiert, das heißt, die Funktion, die Mergesort ausführt, ruft sich selbst wieder auf. Konkret heißt dies, dass die zwei Listen, in die die übergebene Liste geteilt wurde selbst mit Mergesort sortiert werden. Die Abbruchbedingung für diesen rekursiven Vorgang ist dabei, dass die übergebene Liste nur noch einelementig ist, denn dann ist diese Liste trivialer Weise richtig angeordnet. Da jede Liste in endlich vielen Schritten in einelementige Listen geteilt werden kann, ist Mergesort mit dieser Bedingung immer terminiert. Nun steigt man wieder in den Rekursionsebenen auf. Die zuvor zerstückelten Listenfragmente werden nun ineinander sortiert. Auf der obersten Ebene

zum Beispiel hat man dann die vordere und die hintere Hälfte der gesamten Liste bereits richtig angeordnet und diese werden dann noch zusammengefügt.

2.1.2 Eigenschaften und Komplexität

Mergesort ist ein stabiler Sortieralgorithmus, das heißt die Reihenfolge von gleichwertigen Elementen wird nicht verändert. Deswegen eignet sich Mergesort besonders für Anwendungen, bei denen die Stabilität im Vordergrund steht. Dies setzt allerdings voraus, dass das zugrunde liegende ineinander sortieren stabil ist. Dies lässt sich induktiv über die Rekursionsebenen zeigen da bei Ineinandermischen am Ende des Algorithmus die Ordnung beachtet wird. Außerdem ist Mergesort kein *In-place*-Verfahren und braucht zusätzlichen Speicherplatz der Größenordung $\mathcal{O}(n)$ (n bezeichne die Anzahl der zu sortierenden Elemente der Liste). Die Anzahl der Vergleiche um eine beliebige Liste mit n Elementen zu sortieren liegt in der Größenordnung $\mathcal{O}(n \cdot \log_2 n)$, wobei betont sei, dass Best-,Avantage- und Worst-Case die selbe Komplexität haben, was ein Vorteil z.B. gegenüber Quicksort ist, bei dem die Komplexität im Worst-Case $\mathcal{O}(n^2)$ beträgt. Eine weitere wichtige Eigenschaft ist, dass die Ressourcenanforderungen für Mergesort nicht von der anfänglichen Reihenfolge seiner Eingabewerte abhängen, denn die Anzahl der Durchläufe hängt nur von n ab, und nicht von den Elementen selbst. Lediglich die Anzahl der Vergleiche (und damit auch die Prozesszeit) verändert sich mit der Änderung der Reihenfolge. [1].

2.1.3 Experimente

In der Theorie haben wir bereits einige wichtige Eigenschaften von Mergesort gesehen. Nun wollen wir unseren Algorithmus experimentell untersuchen. In Tabelle 2.1 sind Prozesszeit sowie Anzahl der Vergleiche aufgelistet, wobei das eingelesene Textdokument jeweils nur aus "a b a b..." besteht. Dabei bezeichnet n die Anzahl der Wörter. Im Abbildung 2.2 sind Anzahl der Wörter und Anzahl der Vergleiche nochmal in ei-

n	Anzahl Vergleiche	Prozesszeit in ms
2^1	1	0.05
2^2	5	0.08
2^3	12	0.12
2^4	44	0.12
2^5	112	0.18
2^6	272	0.3
2^7	640	0.64
2^8	1472	1.31
2^9	3328	2.67
2^{10}	7424	4.31

Tabelle 2.1: Analyse von Mergesort

[1] Vgl. für diesen Abschnitt [3], S. 350-371

nem Diagramm abgetragen. Die Kurve sieht linear aus, was auf eine gut zu sortierende Wortmenge schließen lässt (dennoch behalten wir im Kopf, dass selbst im Best-Case die Komplexität in $\mathcal{O}(n \cdot \log_2 n)$ liegt). Die Stabilität wurde bereits exemplarisch am Beispieldurchlauf (1.2) untersucht.

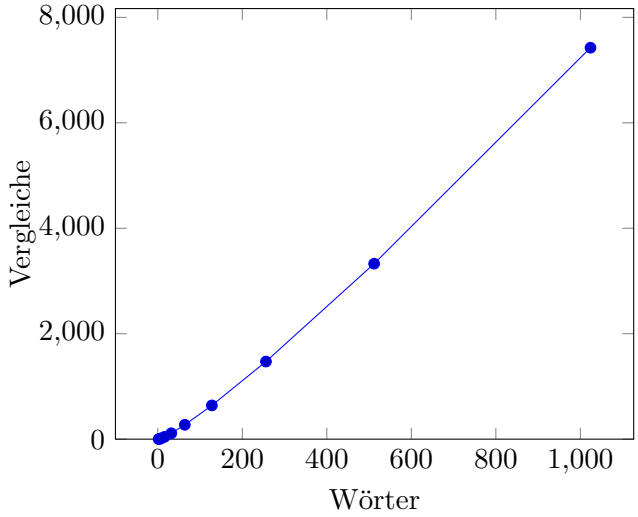

Abbildung 2.2: Daten zu Mergesort

2.2 Combsort

2.2.1 Algorithmus

Combsort ist eine Modifizierung vom Sortierverfahren *Bubblesort*, bei dem die Elemente in der Liste jeweils mit ihrem Nachfolger verglichen werden. Durch die Einführung der Variable `gap`, welche immer kleinere Werte annimmt, wird die Liste derart vorsortiert, dass wenn `gap` 1 annimmt, was dem Bubblesort-Algorithmus entspricht nicht mehr viele Vertauschungen vorgenommen werden müssen. Das Resultat ist eine im Durchschnitt schnellere Laufzeit, was ob der geringen Komplexität der Anpassung gegenüber Bubblesort recht beachtlich ist.

2.2.2 Eigenschaften und Komplexität

Im Beispieldurchlauf zu `combsort(word_list)` hat man gesehen, dass der Algorithmus nicht stabil läuft, da die ursprüngliche Reihenfolge der beiden Elemente "Opel“ und "Opel“ mit dem gleichen Sortierschlüssel nicht eingehalten wurde. Da bei dem Verfahren lediglich Elemente innerhalb der Liste, welche sortiert werden soll, vertauscht werden, wird kein zusätzlicher Speicherplatz benötigt. Das Verfahren arbeitet also *Inplace*. Die Laufzeitkomplexität im Best-case ($\Theta(n \cdot log(n))$ ist besser gegenüber Bubblesort, wohingegen die Laufzeitkomplexität für den Worst-Case mit $\mathcal{O}(n^2)$ die gleiche ist.

Für den Average-case beträgt die Laufzeitkomplexität $\Omega(\frac{n^2}{2^p})$, wobei p der Anzahl der Veränderungen von `gap` entspricht. [2]

2.2.3 Experimente

Zur experimentellen Untersuchung der Laufzeit nutzen wir - wie bereits für Mergesort - Listen, die eine Folge von abwechselnden a's und b's enthalten. Wir ziehen die Summe der durchgeführten Vergleiche und Vertauschungen heran und betrachten außerdem wieder die Prozesszeit in Abhängigkeit von der Anzahl der Wörter. Tabelle 2.2 enthält die erhobenen Daten und Abbildung 2.3 bildet die Summe der Operationen über die Wörteranzahl ab. Auch hier lässt sich wieder eine approximativ lineare Entwicklung

n	Summe Operationen	Prozesszeit in ms
2^1	1	0.0253
2^2	10	0.0314
2^3	26	0.3445
2^4	94	0.0489
2^5	215	0.0712
2^6	596	0.1461
2^7	1432	0.3327
2^8	3611	0.9623
2^9	15081	3.1086
2^{10}	19430	4.1476

Tabelle 2.2: Analyse von Combsort

der Operationen in Abhängigkeit von der Wörteranzahl erkennen, wobei es für 512 (2^9) Wörter eine Abweichung von der Linearität gibt, welche sich eventuell dadurch erklären lässt, dass `gap` ungünstige Werte annimmt, sodass bei niedrigen Werten von `gap` noch vergleichsweise viele Elemente vertauscht werden müssen.

[2] Vgl. für diesen und den vorherigen Abschnitt [1] und [2], S. 315-321

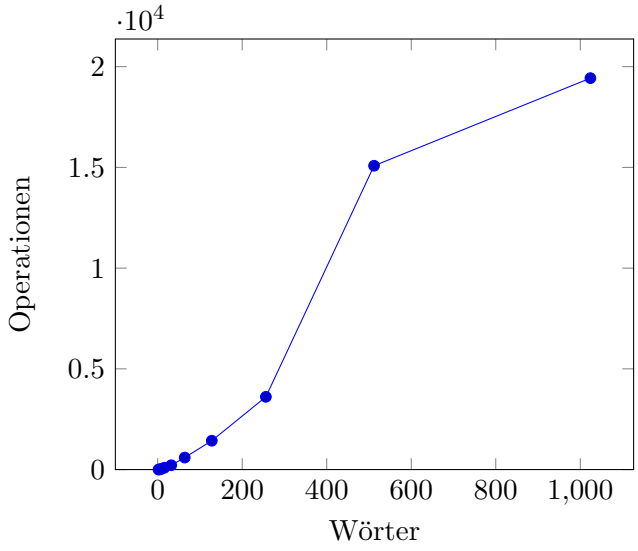

Abbildung 2.3: Daten zu Combsort

3 Zusammenfassung

Es gibt verschiedene Verfahren um eine Liste zu sortieren und zum gleichen Ziel zu kommen. Dabei hängt die Wahl des Verfahrens von zwei wesentlichen Kriterien ab. Als erstes von den gegebenen Systembedingungen. Falls man große Mengen sortieren möchte, aber nur über begrenzten Speicher verfügt, sollte man ein in-place Verfahren wie Combsort implementieren. Allerdings besitzt Combsort im Vergleich zu Mergesort im Worst-Case Szenario eine schlechtere Komplexität und damit auch in der Theorie eine längere Prozessdauer. Das zweite Kriterium das man für die Wahl des richtigen Algorithmus berücksichtigen sollte, ist ob das Ergebnis stabil seien muss. Mergesort beispielsweise ist stabil, während Combsort instabil ist. Unsere Experimente stützen diese Aussagen.

Literaturverzeichnis

[1] Bronislava Brejová. Analyzing variants of shellsort. *Information Processing Letters*, 79(5):223 – 227, 2001.

[2] G. Pomberger and H. Dobler. *Algorithmen und Datenstrukturen: eine systematische Einführung in die Programmierung.* Pearson, 2008.

[3] Robert Sedgewick. *Algorithmen in C++.* Addison-Wesley, Bonn [u.a.], 1993.