**P4.2 Mergesort "almost *in-place*" ★★**

Der Sortieralgorithmus Mergesort der Vorlesung arbeitet nicht *in-place*. Es ist tatsächlich auch nicht möglich, eine *in-place*-Variante zu realisieren.

1. Diskutiert, warum man insbesondere den Merge-Schritt für die beiden Listen [3, 4, 6] und [1, 2, 7] (bzw. andere mögliche dreielementige Listen) nicht *in-place* realisieren kann.
2. In den Hausaufgaben sollt ihr Mergesort "fast *in-place*" realisieren, wobei "nur" der doppelte Speicherplatz benötigt wird. Dabei verwendet man nur eine weitere Liste gleicher Länge wie die Eingabeliste für die Ausgabe bzw. Zwischenergebnisse.

Untersucht dazu als Vorbereitung, wie man die Teillisten, die beim Mergen auftreten, auch mit Hilfe einer einzigen Liste und der Länge der aktuell betrachteten Teillisten darstellen kann. Als Beispiel kann die Zwischenliste

[[1, 4, 6, 8], [2, 3, 4, 6], [5, 7, 9]]

unter Verwendung der Zusatzinformation, dass die Teillisten die Länge 4 haben, als eine eindimensionale Liste dargestellt werden:

[1, 4, 6, 8, 2, 3, 4, 6, 5, 7, 9]

* + Diskutiert, warum diese Darstellung speichertechnisch effizienter ist als eine Liste von Listen.
  + Diskutiert, wie sich nun mit nur einer weiteren Liste, die genauso lang ist wie die Eingabeliste, der Mergesort-Algorithmus umsetzen lässt.
  + Stellt dazu auch den Ablauf beim Sortieren der Liste [4, 1, 6, 8, 6, 4, 2, 3, 9, 5, 7] schrittweise grafisch dar.

**Hinweis:** In den Hausaufgaben nächste Woche sollt ihr diese Überlegungen zur effizienteren "fast *in-place*"-Implementierung von Mergesort nutzen.

**Lösung:**

1. Bei einer *in-place*-Implementierung müssten beide Teillisten in derselben Liste vorgehalten werden, sprich in der Liste [3, 4, 6, 1, 2, 7]. Wenn man die beiden Teillisten nun mergen möchte, bekommt man schon direkt bei dem ersten Wert ein Problem. Die 1 müsste an den Platz der 3 geschrieben werden. Wo soll man also die 3 ablegen? Den Platz der 1 können wir nicht verwenden, da ja die 2 ebenfalls kleiner als die 3 ist. Im schlechtesten Fall sind alle Elemente der ersten Liste größer als die Elemente der zweiten Liste und wir müssten somit die Hälfte aller Elemente der Liste in einer Hilfsliste zwischenspeichern, was eben nicht in-place bedeutet.
2. Diese Darstellung spart Speicherplatz und internen Verwaltungsaufwand, da nicht mehr so viele Listen als Zwischenlisten erstellt werden müssen, sondern nur jeweils eine, die alle Zwischenlisten enthält.

Es reichen in jedem Schritt zwei Listen: Die Liste der aktuellen Zwischenlisten als Eingabeliste und eine Ausgabeliste, in die wir die Liste der gemergten Zwischenlisten schreiben. Die Ausgabeliste ist im nächsten Schritt die Eingabeliste, die alte Eingabeliste wird nicht mehr benötigt und kann als neue Ausgabeliste dienen. Indem wir also die Rollen von Ausgabe- und Eingabeliste in jedem Schritt vertauschen, kommen wir insgesamt mit zwei Listen aus.

Der Ablauf beim Sortieren der Eingabeliste l\_in mit einer weiteren Liste l\_out läuft für das Beispiel wie folgt ab (size gibt hier jeweils die aktuelle Länge der Teillisten an):

zu Beginn:

l\_in = [4, 1, 6, 8, 6, 4, 2, 3, 9, 5, 7], l\_out = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], size= 1

- - - - - - - - - - -

nach Runde 1 und Vertauschen von l\_out und l\_in:

l\_in = [1, 4, 6, 8, 4, 6, 2, 3, 5, 9, 7], l\_out = [4, 1, 6, 8, 6, 4, 2, 3, 9, 5, 7], size = 2

---- ---- ---- ---- ---- -

nach Runde 2 und Vertauschen von l\_out und l\_in:

l\_in = [1, 4, 6, 8, 2, 3, 4, 6, 5, 7, 9], l\_out = [1, 4, 6, 8, 4, 6, 2, 3, 5, 9, 7], size = 4

---------- ---------- -------

nach Runde 3 und Vertauschen von l\_out und l\_in:

l\_in = [1, 2, 3, 4, 4, 6, 6, 8, 5, 7, 9], l\_out = [1, 4, 6, 8, 2, 3, 4, 6, 5, 7, 9], size = 8

---------------------- -------

nach Runde 4 und Vertauschen von l\_out und l\_in:

l\_in = [1, 2, 3, 4, 4, 5, 6, 6, 7, 8, 9], l\_out = [1, 2, 3, 4, 4, 6, 6, 8, 5, 7, 9], size = 11

-------------------------------

Die Eingabeliste l\_in enthält am Ende das Ergebnis. Falls l\_in am Ende nicht auf die ursprüngliche Eingabeliste zeigt, sondern auf die neu erstellte Liste, müssen wir am Ende alle Werte noch einmal alle Werte von l\_in nach l\_out umkopieren.

**P3.2 Quicksort mit gleich großen Elementen ★★**

In der Vorlesung haben wir den Sortieralgorithmus Quicksort kennengelernt und implementiert. Wir haben gesehen, dass auf- oder absteigend sortierte Listen den Worst-Case mit einer Laufzeit von *O*(*n*2)

darstellen (*n*=

Länge der Liste). Diesen Fall haben wir durch Randomisierung extrem unwahrscheinlich gemacht, so dass er in der Praxis kaum noch auftritt. Allerdings gibt es noch einen weiteren Worst-Case-Fall, welcher mittels Randomisierung nicht eliminiert werden kann: Es handelt sich um den Fall, dass alle (bzw. sehr viele) Elemente der Liste gleich groß sind.

1. Analysiert den Fall und auch, warum Randomisierung des Pivot-Elementes hier keine Lösung darstellt.
2. Optimiert die Quicksort-Implementierung in quicksort.py (nicht *in-place*), so dass das häufige Vorkommen identischer Elemente nicht mehr zum Worst-Case führt.
3. Analysiert die Laufzeitkomplexität eurer angepassten Version von Quicksort für eine Liste der Länge *n*
4. , die ausschließlich aus identischen Elementen besteht.

**Lösung:**

1. Angenommen, alle oder sehr viele Listenelemente sind gleich. Teilt man die Liste nun in kleinere und größere Elemente bzgl. des Pivot-Elements auf, so werden alle gleich großen Elemente in dieselbe Liste einsortiert, was daran liegt, dass die Elemente mittels < verglichen werden und sie beim Ergebnis False in die Liste der größeren Elemente einsortiert werden. Auch das Randomisieren stellt hier keine Lösung dar, da sich nach dem Vertauschen des ersten mit einem beliebigen anderen Element (zumindest mit hoher Wahrscheinlich) keine Veränderung in der Aufteilung ergibt.
2. Um diesen Fall zu eliminieren ist es wichtig, dass man nicht die kleineren und die "größer-gleichen", sondern die kleineren, die gleich großen und die größeren Elemente identifiziert. Hierzu passen wir die Funktion split so an, dass sie nun anstelle des Pivot-Elements eine Liste mit dem Pivot-Element und allen identischen Elementen zurück gibt. Der Code steht in der Datei quicksortEqual.py im Anhang.
3. Bei *n*

identischen Elementen wird split nur einmal ausgeführt und gibt zwei leere Listen, sowie eine Liste der Länge *n* mit den Elementen zurück, die gleich groß dem Pivot-Element sind. Welches Element als Pivot-Element verwendet wird, ist dabei egal. Die Laufzeit von split ist linear in *n*.  
Die rekursiven Aufrufe von quicksort mit der leeren Liste benötigen konstante Laufzeit. Die Konkatenation der Ergebnisliste hat wieder lineare Laufzeit in *n*.  
Damit erhalten wir für diesen Fall insgesamt eine lineare Laufzeitkomplexität in *n*

1. , wodurch dieser Fall zum Best-Case wird.

**P3.3 Bucketsort ★ (nur EinfAlg)**

Das Sortierverfahren Bucketsort ist ein Sortierverfahren für Listen von Werten, das besonders gut funktioniert, wenn die Werte möglichst gleichverteilt aus einem begrenzten Wertebereich stammen. Die Grundidee ist folgendermaßen:

* Wir stellen uns den Wertebereich in *k*

disjunkte Bereiche aufgeteilt vor (d. h. wir partionieren den Wertebereich in *k*

* Teile). Jeder Bereich entspricht einem "Bucket" (Eimer) bzw. einer Unterliste.
* Anschließend gehen wir die Liste durch und fügen jeden Wert zu dem Bucket hinzu, zu dessen Wertebereich er gehört.
* Die einzelnen Buckets werden dann jeweils für sich mit einem beliebigen Sortierverfahren sortiert, z. B. mit *in-place*-Quicksort.
* Anschließend schreiben wir der Reihe nach die Werte aus allen Buckets in die ursprüngliche Liste zurück.

1. Skizziert Schritt für Schritt den Ablauf dieses Verfahren zum Sortieren der folgenden Liste von Wörtern mit *k*=26

 Buckets, wobei die Wörter nach ihrem Anfangsbuchstaben a – z auf die Buckets verteilt werden:

['red', 'blue', 'green', 'yellow', 'black', 'white', 'cyan', 'magenta', 'gray', 'brown', 'ruby', 'rose']

 Diskutiert, ob diese Verteilung auf die Buckets zum Sortieren von Listen von Wörtern geeignet ist und macht Vorschläge für alternative Verteilungen für *k* Buckets.

 Erläutert, welche Laufzeitkomplexität das Verfahren im Best-Case und im Worst-Case bezüglich der Länge *n* der Liste hat.

**Hinweis:** In den Hausaufgaben nächste Woche sollt ihr diese Überlegungen zur Implementierung von Bucketsort nutzen.

**Lösung:**

1. Hier werden nur die nicht-leeren Buckets dargestellt.
   * Buckets nach dem Hinzufügen der Wörter:
2. b: ['blue', 'black', 'brown']
3. c: ['cyan']
4. g: ['green', 'gray']
5. m: ['magenta']
6. r: ['red', 'ruby', 'rose']
7. w: ['white']
8. y: ['yellow']
   * Buckets nach dem Sortieren:
9. b: ['black', 'blue', 'brown']
10. c: ['cyan']
11. g: ['gray', 'green']
12. m: ['magenta']
13. r: ['red', 'rose', 'ruby']
14. w: ['white']
15. y: ['yellow']
    * Liste nach dem Zusammenfügen der Buckets:
16. ['black', 'blue', 'brown', 'cyan', 'gray', 'green', 'magenta', 'red', 'rose', 'ruby', 'white', 'yellow']

Für Wörter der Alltagssprache (hier z. B. englische Begriffe) ist die Aufteilung ungünstig, da die Anfangsbuchstaben a – z nicht mit gleicher Häufigkeit vorkommen. Für *k*

 Buckets wäre eine Aufteilung praktikabler, die auf den statistischen Häufigkeiten der Anfangsbuchstaben basiert, so dass im Durchschnitt ähnlich viele Wörter in jedem Bucket erwartet werden.

 Im Best-Case enthalten alle Buckets *n*/*k*

viele Wörter. Der Laufzeitkomplexität für das Sortieren beträgt dann *k*⋅*log*(*n*/*k*)⋅*n*/*k*=*log*(*n*/*k*)⋅*n*, wenn ein Sortierverfahren wie Quicksort verwendet wird, das im Best-Case eine Laufzeitkomplexität von log(*n*)⋅*n* für *n* Elemente hat.

Im Worst-Case werden alle Elemente demselben Bucket zugeordnet. In diesem Fall entspricht die Laufzeitkomplexität genau der Worst-Case-Laufzeit des verwendeten Sortierverfahrens.