**H5.1 Mergesort "almost *in-place*" ★★**

In dieser Aufgabe sollt ihr eine optimierte Variante von Mergesort implementieren, bei der ihr nur zwei Listen verwendet, zwischen denen ihr beim Mischen die Elemente jeweils hin und her kopiert (wie in **Präsenzaufgabe 4.2** diskutiert). Die grundlegende Idee sollte anhand des folgenden Beispiels noch einmal verdeutlicht werden:

| **Eingabeliste** | **Ausgabeliste** | **(bezieht sich jeweils auf das Mischen)** |
| --- | --- | --- |
| [4,6,8,2,5,3,1,7] | [0,0,0,0,0,0,0,0] | Startzustand |
| [4,6,8,2,5,3,1,7] | [4,6,2,8,3,5,1,7] | Mischen der Teillisten der Länge 1 |
| [4,6,2,8,3,5,1,7] | [4,6,8,2,5,3,1,7] | Vertauschen der Listen/ Ausgabeliste steht jetzt vorne |
| [4,6,2,8,3,5,1,7] | [2,4,6,8,1,3,5,7] | Mischen der Teillisten der Länge 2 / Mischen passiert in der Ausgabeliste |
| [2,4,6,8,1,3,5,7] | [4,6,2,8,3,5,1,7] | Vertauschen der Listen |
| [2,4,6,8,1,3,5,7] | [1,2,3,4,5,6,7,8] | Mischen der Teillisten der Länge 4 |

Die Ausgabeliste enthält am Ende das Ergebnis (falls am Ende nicht noch einmal vertauscht wird) und muss nur noch ggf. **in die ursprünglich zu sortierende Liste zurückkopiert werden.**

1. Realisiert zunächst eine neue Variante der merge-Funktion, welche nun mehr Parameter benötigt (mindestens Eingabeliste, Ausgabeliste und Länge der zu mergenden Teillisten in der Eingabeliste, ggf. auch deren Anfangspositionen in der Eingabeliste). Orientiert euch dazu am Beispiel.
2. Anschließend könnt ihr dann das eigentliche Mergesort realisieren. Hierbei sollen nur zwei Listen im Speicher gehalten (die übergebene Liste und eine weitere Liste gleicher Länge), die abwechselnd als Ein- und Ausgabeliste für den Merge-Schritt verwendet werden. 4 P.

**Hinweis:** Vermeidet es, beim Wechsel zwischen den Ein- und Ausgabeliste erneut alle Werte zu kopieren. Das Vertauschen der Referenzen ist aber in Ordnung, da eine Zuweisung einer Listenreferenz zu einer Variablen konstante Laufzeit hat.

**H5.2 Rekursives Mergesort ★**

Die Version von Mergesort, die in der Vorlesung entwickelt wurde (siehe mergesort.py aus **H5.1**), ist iterativ und nicht-mutierend.

Implementiert eine **rekursive, nicht-mutierende** Version von Mergesort als Python-Funktion mergesort\_recursive. Achtet darauf, dass die Funktion in jedem Fall eine neue Liste zurückgibt.

**H5.3 Quicksort *in-place* mit gleich großen Elementen ★★ (nur EinfAlg)**

Auch der in der Vorlesung vorgestellte *in-place*-Quicksort-Algorithmus hat in dem Fall, dass alle (oder sehr viele) Elemente der Liste gleich sind, eine quadratische Laufzeit in der Länge der Liste. Um dies zu umgehen, kann der Algorithmus der Vorlesung so angepasst werden, dass die Elemente in drei Bereiche aufgeteilt werden: die kleineren, die gleich großen und die größeren (jeweils bzgl. des Pivot-Elements). Diese Strategie wurde in Präsenzaufgabe **P3.2** für die Nicht-*in-place*-Version von Quicksort umgesetzt.

Entwickelt eine *in-place*-Version von Quicksort, die diese Strategie ebenfalls umsetzt. Ihr könnt dazu als Grundlage die Implementierung quicksortInPlace.py aus der Vorlesung verwenden und geeignet anpassen.

**Tipp:**

Die bisherige Idee der *in-place*-Implementierung von Quicksort war, dass wir den Index m als Zeiger auf das Ende der Werte verwendet haben, die kleiner als das Pivot-Element waren. Dann konnten wir durch geschicktes Vertauschen weitere kleinere Elemente hierher tauschen und m inkrementieren, während wir mit dem Index i über die Liste laufen:

Wenn wir nun auch die gleich großen Werte ablegen wollen, ist es sinnvoll, einen weiteren Index k zu verwenden, um die folgende Aufteilung zu erzeugen:

**H5.4 Sortieren mit lexikographischer Ordnung ★ (nur EinfAlg)**

Zahlen und Strings (Zeichenketten) werden üblicherweise unterschiedlich sortiert. Zahlen werden numerisch verglichen, während für Strings die [lexikographische Ordnung](https://de.wikipedia.org/wiki/Lexikographische_Ordnung) verwendet wird: Ein String *s*1

ist kleiner als ein String *s*2, wenn entweder *s*1 der Anfang von *s*2 ist, *s*1 mit einem kleineren Zeichen anfängt als *s*2 oder *s*1 und *s*2 zwar den selben Anfang haben, danach aber in *s*2 ein größeres Zeichen folgt als in *s*1.

**Zum Beispiel:** *haus* < *haustier*

(Anfang), *haustier* < *maus* (*h*<*m*), *hausaufgabe* < *haustier* (nach dem gemeinsamen Anfang *haus* haben wir *a*<*t*).

Die lexikographische Ordnung kann auch auf Zahlen definiert werden, wenn diese wie Strings betrachtet werden. Wir beschränken uns hier auf nicht-negative Zahlen.

…

1. Sortiert die Liste [3, 17, 24, 2, 51] lexikographisch und gebt das Ergebnis an.
2. Gebt eine Liste mit mindestens fünf Zahlen an, bei denen die lexikographische Sortierung und die numerische Sortierung dasselbe Ergebnis liefern. Eure Liste soll sowohl ein- als auch mehrstellige Zahlen enthalten.
3. Implementiert in Python eine Funktion isLexSmaller, die zwei Ganzzahlen als Parameter erwartet und True zurückgibt, wenn die erste Zahl bzgl. der lexikographischen Ordnung kleiner ist als die zweite, sonst False.
4. Nehmt einen Sortieralgorithmus eurer Wahl aus der Vorlesung und verändert ihn so, dass er Listen von Zahlen lexikographisch sortiert.