9 Suchen und Sortieren

Suchen und Sortieren sind zwei elementare Themen der Informatik im Bereich der Algorithmen und Datenstrukturen. Die Python-Standardbibliothek setzt beide mit effizienten Implementierungen um und nimmt einem dadurch viel Arbeit ab. Nichtsdestotrotz ist das Verständnis der zugrunde liegenden Algorithmen hilfreich, um die am besten passende Variante für einen Anwendungsfall wählen zu können. Das Thema Suchen stelle ich hier nur einführend vor. In diesem Kapitel widmen wir uns vorrangig einigen essenziellen Sortierverfahren, weil dabei einige algorithmische Tricks gelernt werden können.

9.1 Einführung Suchen

Beim Verwalten von Daten muss man immer mal wieder auch nach Elementen suchen, etwa nach Kunden mit dem Vornamen »Carsten« oder nach einer Rechnung zu einem bestimmten Bestelldatum. Praktischerweise besitzen alle Container diverse Funktionen, etwa solche, mit denen man nach Elementen suchen kann.

Suchen mit `in()`, `index()` und `count()`

Manchmal muss man lediglich prüfen, ob gewisse Daten in einem Container enthalten sind. Dabei hilft das Schlüsselwort `in`. Teilweise möchte man aber auch die Position des Elements erhalten. Dann nutzt man `index()`, das jedoch einen `ValueError` bei Nichtexistenz auslöst:

```
3 in [1, 2, 3] # => True

[1, 2, 3].index(2) # => 1
[1, 2, 3].index(4) # => ValueError
```

Darüber hinaus gilt es zu bedenken, dass `index()` immer den Index des ersten Auftretens liefert. Gibt es mehrere gleiche Elemente und sollen auch alle deren Fundstellen ermittelt werden, dann kann man sich mit einer Kombination aus `in` und `enumerate()` helfen. Bei Bedarf liefert `count()` die Anzahl der gleichen Elemente:

```
print([1, 2, 3, 2].index(2)) # => 1

print([i for i, val in enumerate([1, 2, 3, 2, 4, 2]) if val == 2]) # => [1, 3, 5]
print([1, 2, 3, 2, 4, 2].count(2)) # => 3
```

Für Dictionaries kann man die Schlüssel (`keys()`), die Werte (`values()`) oder deren Kombination mit `items()` ermitteln. Bei Abfragen gibt es für die Schlüssel noch eine Kurzform:

```
programmers = {"Michael": "Python",
               "Tim": "C++",
               "Karthi": "Java"}

if "Karthi" in programmers.keys():
    print("Karthi is here")

# Python-Kurzform
if "Karthi" in programmers:
    print("Karthi is here II")

if "C++" in programmers.values():
    print("someone knows C++")

if ("Michael", "Python") in programmers.items():
    print("Michael knows Python")
```

Darüber hinaus kann man mit `all()` prüfen, ob eine Menge von Elementen enthalten ist. Mit `any()` lässt sich ermitteln, ob es überhaupt eine Übereinstimmung gibt:

```
print(all([7, 2] in [2, 3, 5, 7, 9])) # => True
print(any([7, 2] in [2, 3, 5, 7, 9])) # => True
print(any([4] in [2, 3, 5, 7, 9])) # => False

# Alternative mit mehr Flexibilität
print(all(elem in [2, 3, 5, 7, 9] for elem in [7, 2])) # => True
print(any(elem in [2, 3, 5, 7, 9] for elem in [7, 2])) # => True
print(any(elem in [2, 3, 5, 7, 9] for elem in [4])) # => False
```

Suchen mit `rindex()` und `rfind()`

Für Strings gibt es die Funktionen `rindex()` und `rfind()`, um die Position eines gesuchten Elements vom Ende der Zeichenkette zu ermitteln:

```
print("Hallo".rindex("l")) # => 3
print("Hallo".rfind("l")) # => 3
print("Hallo".rfind("x")) # => -1
# print("Hallo".rindex("x")) # => ValueError: substring not found
```

Leider existiert Derartiges nicht für Listen. Allerdings kann man das recht einfach selbst programmieren, wobei ich hier den aus Java bekannten und meiner Ansicht nach besser verständlichen Funktionsnamen `last_index_of()` nutze. Wird kein Element gefunden, so ist die Rückgabe -1:

```
def last_index_of(values, search_for):
    for pos in range(len(values) - 1, -1, -1):
        if values[pos] == search_for:
            return pos

    return -1


print(last_index_of([1, 2, 3, 2, 4, 2, 5, 2], 2))  # => 7
```

9.1.1 Binärsuche

Neben den gerade genannten Suchfunktionen, die iterativ so lange alle Elemente der Datenstruktur betrachten, bis sie fündig geworden sind, kann man eine effiziente Suche, die sogenannte ***Binärsuche***, anbieten. ***Diese setzt allerdings zwingend sortierte Daten voraus***. Wenn man Daten explizit erst sortieren muss, dann ist der Vorteil gegenüber einer linearen Suche gerade bei kleinen Datenbeständen kaum gegeben.

Bei größeren Datenvolumina ist die logarithmische Laufzeit einer Binärsuche allerdings deutlich besser als diejenige einer linearen Suche: Die geringe Laufzeit wird dadurch erreicht, dass der Algorithmus die jeweils zu verarbeitenden Bereiche halbiert und danach im passenden Teilstück weitersucht. Abbildung 9-1 stellt den prinzipiellen Ablauf dar, wobei aussortierte Teilstücke grau markiert sind.

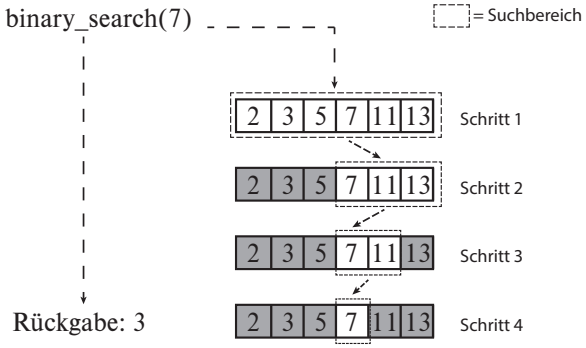

Abbildung 9-1 *Schematischer Ablauf bei der Binärsuche*

In der Abbildung zeigt der Pfeil im ersten Schritt zwischen die Elemente – je nach Implementierung von `binary_search()` wird bei gerader Anzahl das zur Mitte direkt benachbarte linke bzw. rechte Element zum Vergleich genutzt.

9.2 Einführung Sortieren

In diesem Abschnitt stelle ich Ihnen einige Sortieralgorithmen vor, die die Grundlage für die späteren Übungsaufgaben bilden.

9.2.1 Insertion Sort

Insertion Sort kann man sich am besten mit dem Sortieren der eigenen Karten in der Hand bei einem Kartenspiel vorstellen. Man beginnt auf der linken Seite, nimmt dann immer die nächste rechts benachbarte Karte und fügt diese passend in den bereits sortierten linken Teil ein, wodurch in der Regel einige der bereits sortierten Karten nach rechts wandern. Bei diesem Vorgehen kann man die erste Karte überspringen, da sie für sich betrachtet sortiert ist und somit mit der zweiten Karte starten. Schauen wir uns das für die Zahlenfolge 4, 2, 7, 9, 1 an — dazu ist das jeweilige neu einzusortierende Element markiert und der links bereits sortierte Teil vom unsortierten Teil rechts mit || abgetrennt:

```
4 || ② 7 9 1

2 4 || ⑦ 9 1

2 4 7 || ⑨ 1

2 4 7 9 || ①

1 2 4 7 9
```

Im Beispiel startet man mit dem Wert 2. Für jede Zahl muss man die korrekte Einfügeposition ermitteln. Dafür gibt es die nachfolgend beschriebenen zwei Varianten.

Einfügeposition bestimmen

Ausgehend von der aktuellen Position läuft man so lange nach links, wie die Vergleichswerte größer sind. Alternativ kann man auch von vorne beginnen und so lange eine Position nach rechts gehen, wie die verglichenen Werte kleiner sind:

```
def find_insert_pos_from_current(values, current_pos):
    insert_pos = current_pos
    while insert_pos > 0 and values[insert_pos - 1] > values[current_pos]:
        insert_pos -= 1

    return insert_pos


def find_insert_pos_from_start(values, current_pos):
    insert_pos = 0
    while insert_pos < current_pos and values[insert_pos] < values[current_pos]:
        insert_pos += 1

    return insert_pos
```

Tipp: Stabiles Sortieren

Wenn beim Sortieren für Elemente gleichen Werts deren ursprüngliche Reihenfolge in der Collection bestehen bleibt, dann spricht man von ***stabilem Sortieren***. Das ist oftmals eine zu bevorzugende Eigenschaft, weil ggf. mit den Elementen verknüpfte Daten so nicht in der Reihenfolge durcheinanderkommen.

Für das Beispiel führt die Variante `find_insert_pos_from_current()` zu einem stabilen Sortierverfahren, die zweite dagegen nicht. Ersetzt man dort jedoch das < durch <=, so wird das resultierende Sortierverfahren ebenfalls stabil:

```
while insert_pos < current_pos and \
      values[insert_pos] <= values[current_pos]:
```

Das liegt daran, dass dadurch ein zuletzt gefundenes gleiches Element immer hinter allen gleichen Elementen angeordnet wird.

Sortieren durch Einfügen

Nachdem man die korrekte Einfügeposition für einen Wert kennt, müssen noch alle Werte (bis zur Position des aktuell betrachteten Werts) um eine Position nach rechts geschoben werden. Schließlich wird der Wert an der gefundenen Position eingefügt:

```
def insertion_sort(values):
    for current_pos in range(1, len(numbers)):
        current_val = numbers[current_pos]
        insert_pos = find_insert_pos_from_current(values, current_pos)

        move_right(values, current_pos, insert_pos)

        numbers[insert_pos] = current_val


def move_right(values, current_pos, insert_pos):
    move_pos = current_pos
    while move_pos > insert_pos:
        values[move_pos] = values[move_pos - 1]
        move_pos -= 1
```

Im Listing ist eine gut nachvollziehbare Realisierung gezeigt, die auf Verständlichkeit und nicht auf Geschwindigkeit setzt. Tatsächlich kann man einige Aktionen geschickt miteinander verbinden und so die mehrfachen Durchläufe vermeiden. In der Übungsaufgabe 4 geht es später um genau diese Optimierung.

9.2.2 Selection Sort

Selection Sort ist ein weiteres intuitiv verständliches Verfahren mit zwei Varianten: Eine basiert auf dem Minimum und die andere auf dem Maximum. Bei der Minimum-Variante wird die zu sortierende Liste bzw. das Array von vorne nach hinten durchlaufen und dabei in jedem Schritt das Minimum aus dem noch unsortierten Teilbereich ermittelt. Dieses wird nach vorne bewegt, indem es mit dem aktuellen Element vertauscht wird. Dadurch wächst der sortierte Bereich von vorne und es verkleinert sich der verbliebene noch unsortierte Bereich. Für die Variante mit Maximum durchläuft man die zu sortierenden Daten von hinten nach vorne, ordnet das jeweilige Maximum hinten ein, sodass der sortierte Bereich von hinten wächst.

Zum besseren Verständnis betrachten wir das für ein paar Werte — dazu ist das jeweilige Minimum bzw. Maximum speziell markiert und der sortierte vom unsortierten Teil mit || abgetrennt. Man erkennt gut, wie der sortierte Teil wächst.

```
    MIN                                 MAX
    ->                                   <-
    || 4 2 7 9 ①                4 2 7 ⑨ 1 ||

1:  1 || ② 7 9 4                4 2 ⑦ 1 || 9

2:  1 2 || 7 9 ④                ④ 2 1 || 7 9

3:  1 2 4 || 9 ⑦                1 ② || 4 7 9

4:  1 2 4 7 || 9                1 || 2 4 7 9
```

Nachfolgend ist die Implementierung der Variante für das Minimum gezeigt:

```
def selection_sort_min(values):
    for i in range(len(values) - 1):
        min_idx = i

        # Finde Minimum
        for j in range(i + 1, len(values)):
            if values[j] < values[min_idx]:
                min_idx = j

        # Tausche aktuellen Wert mit Minimum
        tmp = values[min_idx]
        values[min_idx] = values[i]
        values[i] = tmp
```

Wenn man Algorithmen nur auf dieser Low-Level-Ebene anschaut, fällt meistens das Verständnis und Nachvollziehen schwer. Natürlich müssen die endgültigen, in Frameworks eingesetzten Algorithmen möglichst optimal sein. Dazu benötigt man Abschätzungen mit der O-Notation, die auf der Low-Level-Ebene einfacher durchführbar sind als auf der High-Level-Ebene, da dann sämtliche Konstrukte inklusive aufgerufener Funktionen betrachtet werden müssen. Jedoch ist es zum Lernen und zum Einstieg viel geeigneter, zunächst verständlich zu programmieren und dann in weiteren Schritten an der Optimierung zu arbeiten.

Meinung: Starte mit Verständlichkeit

Wie lässt sich Selection Sort auf höherer Abstraktionsebene beschreiben? Dazu greifen wir auf einige Hilfsfunktionen zurück, die wir zu Arrays erstellt haben: Zum einen in der Einführung die Funktion `swap()` zum Vertauschen von Elementen, zum anderen die Funktion `find_min_pos()` zum Auffinden der Position des kleinsten Elements, was als Übungsaufgabe 11 in Abschnitt 6.3.11 ab Seite 287 erstellt wurde. Praktischerweise lassen sich diese ohne Änderung auch für Listen nutzen.

Durch deren Verwendung wird der eigentliche Ablauf nahezu sofort ersichtlich: Wir durchlaufen die Liste von vorne und suchen jeweils das Minimum des verbleibenden Teils und tauschen dieses mit dem Wert der aktuellen Position:

```
def selection_sort_min_readable(values):
    for cur_idx in range(len(values) - 1):
        min_idx = find_min_pos(values, cur_idx, len(values))
        swap(values, min_idx, cur_idx)
```

Im Listing kommen die zwei fett markierten Hilfsmethoden aus der Klasse `ArrayUtils` zum Einsatz. Wie üblich bündelt die Klasse `ArrayUtils` verschiedene in Kapitel 6 entwickelte Hilfsmethoden. Zum leichteren Ausprobieren des Beispiels mit der Kommandozeile sind nachfolgend die zwei im obigen Listing aufgerufenen Hilfsmethoden nochmals abgebildet:

```
def find_min_pos(values, start_pos, end_pos):
    min_pos = start_pos
    for i in range(start_pos + 1, end_pos):
        if values[i] < values[min_pos]:
            min_pos = i

    return min_pos


def swap(values, pos1, pos2):
    temp = values[pos1]
    values[pos1] = values[pos2]
    values[pos2] = temp
```

Probieren wir das mal auf der Kommandozeile aus:

```
>>> values = [ 4, 2, 7, 9, 1 ]
>>> selection_sort_min_readable(values)
>>> print(values)
[1, 2, 4, 7, 9]
```

9.2.3 Merge Sort

Merge Sort basiert auf einem Divide-and-Conquer-Ansatz und zerteilt rekursiv die zu sortierende Liste in immer kleiner werdende Teile mit etwa der Hälfte der ursprünglichen Größe, bis diese nur noch aus einem oder ggf. keinem Element bestehen. Danach werden die Teile wieder vereint. In diesem Merge-Schritt erfolgt das Sortieren durch die passende, auf den jeweiligen Werten basierende Zusammenführung. Die Abläufe kann man sich wie folgt verdeutlichen:

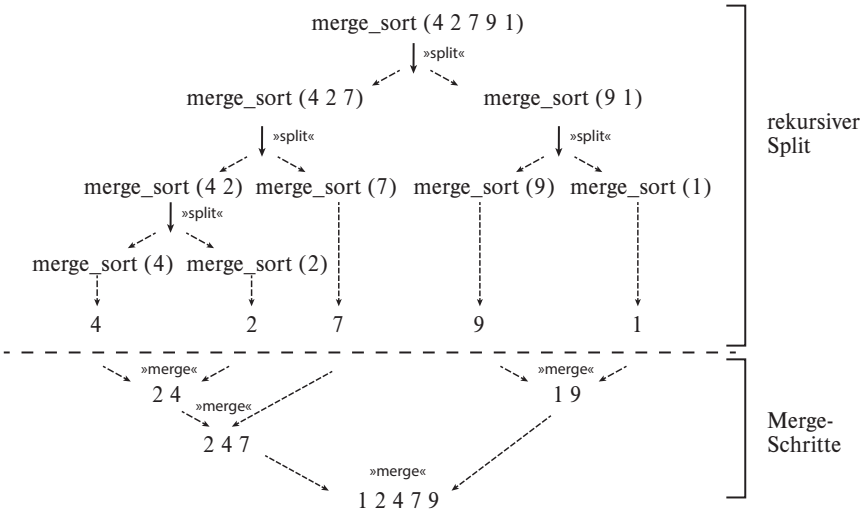

Abbildung 9-2 *Ablauf bei Merge Sort*

Der Algorithmus des Aufsplittens lässt sich rekursiv und gut verständlich – allerdings auch etwas ineffizient – implementieren, sofern man neue Listen erzeugen darf. Dabei kommt die Realisierung der Funktion `merge(values1, values2)` zum Einsatz, die bereits in der Übungsaufgabe 12 in Abschnitt 5.3.12 ab Seite 204 erstellt wurde:

```
def merge_sort(to_sort):
    # Rekursiver Abbruch: Länge 0 (nur wenn initial leere Liste) oder 1
    if len(to_sort) <= 1:
        return to_sort

    # Rekursiver Abstieg: teile in zwei Hälften
    mid_pos = len(to_sort) // 2
    left = to_sort[0: mid_pos]
    result_left = merge_sort(left)

    right = to_sort[mid_pos: len(to_sort)]
    result_right = merge_sort(right)

    # Verbinde die Teilresultate zu größerem sortiertem Datenbestand
    return merge(result_left, result_right)
```

Tipp: Analogie aus dem realen Leben führt zu Optimierung

Auch für Merge Sort lässt sich wieder die Analogie zum Sortieren von Karten eines Kartenspiels nutzen. Wenn man einen ziemlich großen Stapel an Karten sortieren muss, kann man diesen in viele, deutlich kleinere Stapel aufteilen, diese für sich sortieren und danach sukzessive vereinigen. Statt jedoch die Stapel bis auf eine Karte zu reduzieren, bietet es sich an, die kleineren Stapel mit einem anderen Verfahren zu sortieren, oftmals mit Insertion Sort, das für kleine, idealerweise fast geordnete Werte eine Laufzeit von $O(n)$ besitzt. Das kann man zum Feintunen verwenden. Genialerweise ist dies bei Merge Sort kinderleicht:

```
def merge_sort_with_insertion_sort(to_sort):
    # Rekursiver Abbruch inklusive Mini-Optimierung
    if len(to_sort) < 5:
        insertion_sort(to_sort)
        return to_sort

    # Rekursiver Abstieg: teile in zwei Hälften
    mid_pos = len(to_sort) // 2
    left = to_sort[0: mid_pos]
    result_left = merge_sort(left)

    right = to_sort[mid_pos: len(to_sort)]
    result_right = merge_sort(right)

    # Verbinde die Teilresultate zu größerer sortierter Liste
    return merge(result_left, result_right)
```

Abschließend möchte ich noch darauf hinweisen, dass die Grenze, ab wann man auf Insertion Sort wechseln sollte, hier recht willkürlich auf den Wert 5 festgesetzt wurde. Vermutlich sind Werte zwischen 10 und 20 Elemente ziemlich praxistauglich, jedoch sollte man dann auf das Wissen von Algorithmen-Profis zurückgreifen, die für Laufzeiten mathematisch fundierte Abschätzungen erstellen.

9.2.4 Quick Sort

Ebenso wie Merge Sort basiert auch Quick Sort auf einem Divide-and-Conquer-Ansatz und zerteilt den zu sortierenden Datenbestand in immer kleiner werdende Teile. Dabei wird ein spezielles Element (***Pivot*** genannt) ausgewählt, das die Unterteilung festlegt. Der Einfachheit halber kann man das erste Element des zu sortierenden Teilbereichs als Pivot-Element wählen – andere Varianten sind aber denkbar. Bei Quick Sort erfolgt das Sortieren basierend auf diesem Pivot-Element, indem man alle Elemente des Teilbereichs gemäß ihrem Wert links (kleiner oder gleich) bzw. rechts (größer) vom Pivot anordnet. Dadurch ist das Pivot-Element an der korrekten Stelle einsortiert. Nun wird das Ganze rekursiv für die linken und rechten Teile wiederholt, bis die Teilbereiche nur noch aus einem Element bestehen. Die Abläufe zeigt folgende Abbildung.

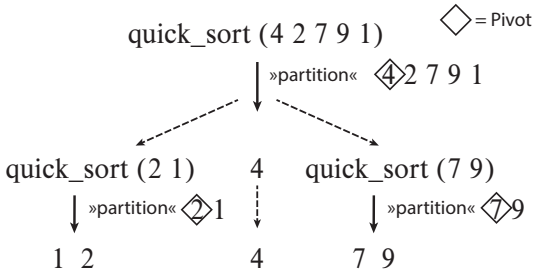

Abbildung 9-3 *Ablauf bei Quick Sort*

Beginnen wir mit einer Umsetzung mit Listen als Hilfsdatenstruktur, da dies leichter nachvollziehbar ist. Dadurch lassen sich das Aufgliedern in kleinere und gleiche sowie größere Elemente einfach implementieren und auch das spätere Zusammenfügen der Ergebnisse der rekursiven Teilberechnungen ist leicht. Die gesamte Implementierung ist bewusst nicht auf Geschwindigkeit, sondern Verständlichkeit optimiert.

Zur Partitionierung sammelt man in je einer Liste alle Elemente, die kleiner oder gleich bzw. die größer als der Wert des Pivot-Elements sind. Dabei überspringen wir das erste Element, da dieses das Pivot-Element ist, und wenden dann die jeweils passende als Lambda übergebene Filterbedingung an.

```
def quick_sort(values):
    # rekursiver Abbruch
    if len(values) <= 1:
        return values

    # Aufsammeln kleiner gleich / größer als Pivot
    pivot = values[0]
    below_or_equals = [value for value in values[1:] if value <= pivot]
    aboves = [value for value in values[1:] if value > pivot]

    # rekursiver Abstieg
    sorted_lowers_part = quick_sort(below_or_equals)
    sorted_uppers_part = quick_sort(aboves)

    # Zusammenfügen
    return sorted_lowers_part + [pivot] + sorted_uppers_part
```

Mit eigenen Listen als Zwischenspeicher und Hilfsdatenstruktur sowie nicht auf Performance optimiert ist das Ganze recht intuitiv. Deutlich unhandlicher wird es, wenn man die Partitionierung für Listen inplace, also direkt in der Originalliste selbst, realisieren möchte. Davon dürfen Sie sich später in Form der Übungsaufgabe 5 überzeugen. Den prinzipiellen Ablauf schauen wir uns nun an.

Inplace-Umsetzung

Der grundsätzliche Algorithmus lässt sich wie folgt implementieren, wobei die Realisierung der Partitionierung – wie schon erwähnt – eine Übungsaufgabe sein wird:

```
def quick_sort_inplace(values):
    quick_sort_in_range(values, 0, len(values) - 1)


def quick_sort_in_range(values, left, right):
    # rekursiver Abbruch
    if left >= right:
        return

    partition_index = partition(values, left, right)

    # rekursiver Abstieg
    quick_sort_in_range(values, left, partition_index - 1)
    quick_sort_in_range(values, partition_index + 1, right)
```

Tipp: Vermeidung von Seiteneffekten durch Kopie

Soll der ursprüngliche Datenbestand nicht verändert werden, so kann man zunächst eine Kopie davon erstellen und danach dann die Inplace-Funktion aufrufen:

```
def quick_sort_with_copy(values):
    copied_values = values.copy()
    quick_sort_inplace(copied_values);

    return copied_values
```

9.2.5 Bucket Sort

Bucket Sort ist ein interessantes Sortierverfahren, dessen Algorithmus ich nachfolgend nur skizziere, da die Implementierung Thema der Übungsaufgabe 7 ist.

Bucket Sort ist ein zweistufiges Verfahren zum Sortieren der Daten. Zunächst werden die Werte in speziellen Containern (den sogenannten Buckets) gesammelt. Danach werden diese Werte dann passend in eine sortierte Liste überführt. Damit der Algorithmus möglich ist, müssen die zu sortierenden Elemente eine begrenzte Wertemenge besitzen. Das gilt etwa für Altersangaben von Personen, wo wir einen Wertebereich von 0 bis 150 annehmen können.

```
ages = [10, 50, 22, 7, 42, 111, 50, 7]
```

Durch diese Festlegung auf eine maximale Anzahl an unterschiedlichen Werten lassen sich korrespondierend viele Behälter, die Buckets, nutzen, um die Werte bzw. genauer deren Häufigkeit zu speichern. Für jeden möglichen Wert wird ein Bucket vorgesehen.

Schritt 1: Verteilung auf Buckets Nun wird die Ausgangsmenge an Daten durchlaufen und deren Auftreten in den Buckets vermerkt. Exemplarisch für die obigen Altersangaben ergibt sich folgende Verteilung:

```
bucket[7] = 2
bucket[10] = 1
bucket[22] = 1
bucket[42] = 1
bucket[50] = 2
bucket[111] = 1
```

Schritt 2: Aufbereitung des sortierten Ergebnisses In einem letzten Schritt werden die Buckets von vorne durchlaufen und die jeweiligen Werte in das Resultat so oft eingefügt, wie deren Anzahl im Bucket hinterlegt ist. Damit entsteht diese Sortierung:

```
result = [7, 7, 10, 22, 42, 50, 50, 111]
```

9.2.6 Schlussgedanken

Viele der intuitiv gut zu erfassenden Algorithmen wie Insertion Sort und Selection Sort haben allerdings den Nachteil, dass sie in der Regel eine Laufzeit von $O(n^2)$ besitzen. Insertion Sort verfügt jedoch über eine positive und bemerkenswerte Besonderheit, sofern die Ausgangsdaten (nahezu) sortiert vorliegen: Dann wird Insertion Sort mit $O(n)$ extrem performant.

Quick Sort und Merge Sort sind in der Regel mit einer Laufzeit von $O(n \cdot log(n))$ sehr performant, allerdings besitzen diese auch eine höhere Komplexität des Sourcecodes, insbesondere, wenn dort inplace gearbeitet wird. Das Inplace-Arbeiten ist für Frameworks und größere Datenbestände sehr wichtig für die Performance. Bei Straight-Forward-Implementierungen von Merge Sort und Quick Sort werden oftmals viele Kopien von Teilbereichen erzeugt (bei Quick Sort bei der Partionierung). Für beides gibt es aber Varianten, die die Aktionen inplace erledigen. Interessanterweise sind die jeweiligen Teilungen der zu sortierenden Teilbereiche per Rekursion ziemlich einfach zu formulieren, jedoch ist der Teil zur Partitionierung bzw. zum Mergen dann komplexer und schwieriger zu implementieren, vor allem, wenn man inplace arbeitet. Für Merge Sort finden Sie ein Beispiel im mitgelieferten Python-Projekt, für Quick Sort dürfen Sie sich in der Übungsaufgabe 6 daran versuchen.

Verbleibt noch Bucket Sort. Dieser Algorithmus läuft mitunter sogar in linearer Laufzeit, allerdings ist er im Gegensatz zu den anderen vorgestellten Sortierverfahren nicht allgemeingültig, da er die schon genannte Restriktion der Einschränkung der möglichen Wertemenge besitzt.

9.3 Aufgaben

9.3.1 Aufgabe 1: Contains All (★★☆☆☆)

Es soll eine Funktion `contains_all(values, search_values)` für Listen erstellt werden, die prüft, ob alle übergebenen Werte in der Ausgangsliste auch vorhanden sind. Benutzen Sie explizit nicht die Python-Standardfunktionalität von `all()`, sondern programmieren Sie dies selbst.

Beispiele

Eingabe	Suchwerte	Resultat
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]	[7, 2]	True
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]	[5, 11]	False

9.3.2 Aufgabe 2: Partitionierung (★★★★☆)

Die Aufgabenstellung besteht darin, eine gemischte Folge der Buchstaben A und B in einem einzigen Durchlauf passend zu sortieren bzw. so anzuordnen, dass alle As vor den Bs auftreten. Das kann man auch auf drei Buchstaben ausweiten.

Beispiele

Eingabe	Resultat
"ABAABBBAAABBBA"	"AAAAAAABBBBBBB"
"ABACCBBCAACCBBA"	"AAAAABBBBBCCCCC"

Aufgabe 2a: Partitionierung von zwei Buchstaben (★★★☆☆)

Schreiben Sie eine Funktion `partition2(text)`, die eine gegebene Folge der zwei Buchstaben A und B in die geordnete Folge bringt, bei der alle As vor den Bs auftreten.

Aufgabe 2b: Partitionierung von drei Buchstaben (★★★★☆)

Schreiben Sie eine Funktion `partition3(text)`, die eine gegebene Folge von drei Buchstaben A, B und C in die geordnete Folge bringt, bei der alle As vor den Bs und diese wiederum vor den Cs auftreten. Statt mit Buchstaben kann man sich dies für Farben einer Flagge vorstellen. Dann ist es unter dem Namen »***Dutch Flag Problem***« bekannt.

9.3.3 Aufgabe 3: Binärsuche (★★☆☆☆)

Aufgabe 3a: Binärsuche rekursiv (★★☆☆☆)

Schreiben Sie eine rekursive Funktion `binary_search(values, search_for)`, die in einer sortierten Liste nach dem gewünschten Wert sucht.

Beispiele

Eingabe	Suchwert	Resultat
[1, 2, 3, 4, 5, 7, 8, 9]	5	True
[1, 2, 3, 4, 5, 7, 8, 9]	6	False

Aufgabe 3b: Binärsuche iterativ (★★☆☆☆)

Wandeln Sie die rekursive Funktion in eine iterative Funktion mit abgeändertem Rückgabewert und folgendem Namen: `binary_search_iterative(values, search_value)`. Liefern Sie statt `True` bzw. `False` die Position des Suchwerts bzw. -1 für nicht gefunden.

Beispiele

Eingabe	Suchwert	Position
[1, 2, 3, 4, 5, 7, 8, 9]	5	4
[1, 2, 3, 4, 5, 7, 8, 9]	6	-1

9.3.4 Aufgabe 4: Insertion Sort (★★☆☆☆)

In der Einführung in Abschnitt 9.2.1 hatte ich eine vereinfachte, gut nachvollziehbare Realisierung von Insertion Sort gezeigt. In dieser Aufgabenstellung geht es darum, das Ganze zu optimieren, indem das Auffinden der Einfügeposition und das dafür nötige Tauschen und Einfügen in einem Rutsch erfolgen. Schreiben Sie eine optimierte Version von `insertion_sort(values)`.

Beispiel

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

9.3.5 Aufgabe 5: Selection Sort (★★☆☆☆)

Schreiben Sie eine Variante von Selection Sort, die statt des Minimums das Maximum nutzt und folgende Signatur besitzt: `selection_sort_max_inplace(values)`.

Was ist zu modifizieren, damit der Sortieralgorithmus die Originaldaten unverändert lässt und eine neue, sortierte Liste zurückgibt? Schreiben Sie dazu eine Funktion `selection_sort_max_copy(values)`.

Beispiel

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

9.3.6 Aufgabe 6: Quick Sort (★★★☆☆)

Quick Sort habe ich in der Einführung in Abschnitt 9.2.4 beschrieben. Während sich die Implementierung der Aufteilung in zwei Bereiche mit Werten kleiner gleich bzw. größer als das Pivot-Element mithilfe von Listen als Hilfsdatenstrukturen einfach und sehr verständlich realisieren lässt, ist das für Listen als Inplace-Variante herausfordernder. Nun soll die Partitionierung mit der Funktion `partition(values, left, right)` umgesetzt werden. Nachfolgend ist noch einmal der bereits bestehende, aufrufende und inplace arbeitende Sourcecode abgebildet:

```
def quick_sort_inplace(values):
    quick_sort_in_range(values, 0, len(values) - 1)


def quick_sort_in_range(values, left, right):
    # rekursiver Abbruch
    if left >= right:
        return

    partition_index = partition(values, left, right)

    # rekursiver Abstieg
    quick_sort_in_range(values, left, partition_index - 1)
    quick_sort_in_range(values, partition_index + 1, right)
```

Beispiele

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

9.3.7 Aufgabe 7: Bucket Sort (★★☆☆☆)

In Abschnitt 9.2.5 wurde der Algorithmus von Bucket Sort beschrieben. In dieser Aufgabe soll eine Funktion `bucket_sort(values, expected_max)` erstellt werden, die dieses Sortierverfahren für ein Liste von Werten und einen erwarteten Maximalwert implementiert.

Beispiel

Eingabe	Maximalwert	Resultat
[10, 50, 22, 7, 42, 111, 50, 7]	120	[7, 7, 10, 22, 42, 50, 50, 111]

9.3.8 Aufgabe 8: Suche in rotierten Daten (★★★★☆)

In dieser Aufgabenstellung sind in einer gegebenen sortierten Folge von Ganzzahlwerten zwei Suchprobleme zu lösen. Die Herausforderung besteht darin, dass die Werte zwar geordnet, aber in sich rotiert sind, sodass das kleinste Element eben nicht vorne und das größte nicht hinten in den Daten zu finden ist (außer im Spezialfall einer Rotation um 0 Positionen).

Tipp Prüfen Sie unbedingt auch den Spezialfall einer Rotation von 0 bzw. einem Vielfachen der Länge der Liste bzw. des Arrays, was wieder einer Rotation um den Wert 0 entspräche.

Aufgabe 8a: Flankenwechsel effizient (★★★★☆)

Schreiben Sie eine Funktion `find_flank_pos(values)`, die eine in einer gegebenen sortierten Folge von n Ganzzahlwerten, etwa 25, 33, 47, 1, 2, 3, 5, 11, effizient in logarithmischer Zeit, also $O(log(n))$, die Position des Flankenwechsels, bestimmt. Schreiben Sie zwei auf `find_flank_pos(values)` basierende Funktionen `min_value(values)` und `max_value(values)`, die gemäß ihrem Namen das Minimum bzw. Maximum aus der gegebenen sortierten, aber rotierten Wertefolge ermitteln.

Beispiele

Eingabe	Flankenposition	Minimum	Maximum
[25, 33, 47, 1, 2, 3, 5, 11]	3	1	47
[5, 11, 17, 25, 1, 2]	4	1	25
[6, 1, 2, 3, 4, 5]	1	1	6
[1, 2, 3, 4, 5, 6]	0 (Spezialfall)	1	6

Aufgabe 8b: Binärsuche in rotierten Daten (★★★★☆)

Schreiben Sie eine Funktion `binary_search_rotated(values, search_for)`, die in einer gegebenen sortierten Folge von Ganzzahlwerten, etwa der Zahlenfolge 25, 33, 47, 1, 2, 3, 5, 11, effizient nach einem übergebenen Wert sucht und dessen Position oder -1 bei Nichtenthaltensein liefert.

Beispiele

Eingabe	Flankenposition	Suchwert	Ergebnis
[25, 33, 47, 1, 2, 3, 5, 11]	3	47	2
[25, 33, 47, 1, 2, 3, 5, 11]	3	3	5
[25, 33, 47, 1, 2, 3, 5, 11]	3	13	-1
[1, 2, 3, 4, 5, 6, 7]	0 (Spezialfall)	5	4
[1, 2, 3, 4, 5, 6, 7]	0 (Spezialfall)	13	-1

9.4 Lösungen

9.4.1 Lösung 1: Contains All (★★☆☆☆)

Es soll eine Funktion `contains_all(values, search_values)` für Listen erstellt werden, die prüft, ob alle übergebenen Werte in der Ausgangsliste auch vorhanden sind. Benutzen Sie explizit nicht die Python-Standardfunktionalität von `all()`, sondern programmieren Sie dies selbst.

Beispiele

Eingabe	Suchwerte	Resultat
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]	[7, 2]	True
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]	[5, 11]	False

Algorithmus Für unsere Implementierung rufen wir die Prüfung mit `in` wiederholt auf, und zwar für alle zur Prüfung auf Enthaltensein übergebenen Elemente:

```
def contains_all(values, search_values):
    for current in search_values:
        if not current in values:
            return False

    return True
```

Python-Shortcut Das kann man mithilfe von `all()` kompakter schreiben, trotzdem bietet sich wohl eine Hilfsfunktion an, um den aufrufenden Sourcecode möglichst verständlich zu halten:

```
def contains_all_v2(values, search_values):
    return all(elem in values for elem in search_values)
```

Prüfung

Definieren wir eine Liste mit den Zahlen von 0 bis 9 und prüfen, ob dort die Werte 7 und 2 sowie 5 und 11 vorhanden sind:

```
@pytest.mark.parametrize("values, search_values, expected",
                         [([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [7, 2], True),
                          ([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [5, 11], False)])
def test_contains_all(values, search_values, expected):
    assert contains_all(values, search_values) == expected


@pytest.mark.parametrize("values, search_values, expected",
                         [([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [7, 2], True),
                          ([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [5, 11], False)])
def test_contains_all_v2(values, search_values, expected):
    assert contains_all_v2(values, search_values) == expected
```

9.4.2 Lösung 2: Partitionierung (★★★★☆)

Die Aufgabenstellung besteht darin, eine gemischte Folge der Buchstaben A und B in einem einzigen Durchlauf passend zu sortieren bzw. so anzuordnen, dass alle As vor den Bs auftreten. Das kann man auch auf drei Buchstaben ausweiten.

Beispiele

Eingabe	Resultat
"ABAABBBAAABBBA"	"AAAAAAABBBBBBB"
"ABACCBBCAACCBBA"	"AAAAABBBBBCCCCC"

Lösung 2a: Partitionierung von zwei Buchstaben (★★★☆☆)

Schreiben Sie eine Funktion `partition2(text)`, die eine gegebene Folge der zwei Buchstaben A und B in die geordnete Folge bringt, bei der alle As vor den Bs auftreten.

Algorithmus Obwohl man zunächst versucht ist, alle möglichen Positionen miteinander zu vergleichen, existiert doch eine extrem pfiffige und performante Lösung, die die Aufgabenstellung in einem Durchlauf löst. Man arbeitet mit zwei Positionszeigern low und $high$, die die vordere und hintere Position, in diesem Fall des gültigen Bereichs, gegeben durch das hinterste A und das vorderste B, markieren. Dieser Bereich ist initial leer und wächst, bis man am Textende ankommt. Man nutzt folgendes Vorgehen: Beim Auffinden eines As wird die Position der As erhöht. Bei einem B wird dieses nach hinten getauscht und der Positionszeiger der Bs verringert, wodurch sich der bereits korrekt aufgeteilte Bereich ausdehnt.

```
def partition2(char_values):
    low = 0
    high = len(char_values) - 1

    while low <= high:
        if char_values[low] == 'A':
            low += 1
        else:
            swap_positions(char_values, low, high)
            high -= 1
            # low muss bleiben, weil ja theoretisch auch ein
            # B nach vorne getauscht werden kann

    return "".join(char_values)


def swap_positions(list, pos1, pos2):
    list[pos1], list[pos2] = list[pos2], list[pos1]
```

Weil beim Tauschen auch ein B nach vorne gelangen kann, darf der untere Positionszeiger nicht verändert werden. Bei einem der nächsten Schritte wird das B dann wieder nach hinten gelangen. Durch diesen trickreichen Algorithmus ist man in der Lage, in einem einzigen Durchlauf alle As vor den Bs anzuordnen.

Lösung 2b: Partitionierung von drei Buchstaben (★★★★☆)

Schreiben Sie eine Funktion `partition3(text)`, die eine gegebene Folge von drei Buchstaben A, B und C in die geordnete Folge bringt, bei der alle As vor den Bs und diese wiederum vor den Cs auftreten. Statt mit Buchstaben kann man sich dies für Farben einer Flagge vorstellen. Dann ist es unter dem Namen »***Dutch Flag Problem***« bekannt.

Algorithmus Die Erweiterung von zwei auf drei Buchstaben (oder Farben) setzt ähnliche Ideen wie zuvor ein, jedoch mit ein paar weiteren Tricks und Spezialbehandlungen. Man beginnt wieder am Anfang des Arrays, nutzt jedoch die drei Positionsmarker low, mid und $high$. Zu Anfang befinden sich diese für das erste und mittlere Zeichen auf der Position 0, der für $high$ auf der Endposition. Findet man ein 'A', so verschiebt sich die Position für low und mid um eins nach rechts. Zuvor muss noch das letzte Zeichen aus dem unteren Bereich mit dem aktuellen (mittleren) vertauscht werden. Liest man ein 'B', so wird nur die mittlere Position in Richtung Ende verschoben. Ist das aktuelle Zeichen ein 'C', so muss dieses nach hinten getauscht werden. Die Positionsmarkierung für den oberen Bereich wird dann um 1 verringert.

```
def partition3(char_values):
    low = 0
    mid = 0
    high = len(char_values) - 1

    while mid <= high:
        if char_values[mid] == 'A':
            swap_positions(char_values, low, mid)
            low += 1
            mid += 1
        elif char_values[mid] == 'B':
            mid += 1
        else:
            swap_positions(char_values, mid, high)
            high -= 1
            # low und mid müssen bleiben, weil ja theoretisch auch ein B oder C
            # nach vorne getauscht werden kann

    return "".join(char_values)
```

Prüfung

Zum Überprüfen der Funktionalität nutzen wir zwei Strings, bestehend aus einer gemischten Folge der Buchstaben A und B bzw. A, B und C:

```
def test_partition2():
    assert partition2(list("ABAABBBAAABBBA")) == "AAAAAAABBBBBBB"


def test_partition3():
    assert partition3(list("ABACCBBCAACCBBA")) == "AAAAABBBBBCCCCC"
```

9.4.3 Lösung 3: Binärsuche (★★☆☆☆)

Lösung 3a: Binärsuche rekursiv (★★☆☆☆)

Schreiben Sie eine rekursive Funktion `binary_search(values, search_for)`, die in einer sortierten Liste nach dem gewünschten Wert sucht.

Beispiele

Eingabe	Suchwert	Resultat
[1, 2, 3, 4, 5, 7, 8, 9]	5	True
[1, 2, 3, 4, 5, 7, 8, 9]	6	False

Algorithmus Teile die Liste in zwei Hälften. Bestimme den Wert in der Mitte und schaue, ob in der unteren oder oberen Hälfte weiter gesucht werden muss. Das lässt sich aufgrund der gegebenen Sortierung leicht feststellen:

$$\begin{aligned} Wert_{mitte} &== Suchwert &\Rightarrow& \text{ gefunden, Ende} \\ Wert_{mitte} &< Suchwert &\Rightarrow& \text{ oben weiter suchen} \\ Wert_{mitte} &> Suchwert &\Rightarrow& \text{ unten weiter suchen} \end{aligned}$$

Die Umsetzung in Python folgt strikt der Beschreibung – wie üblich muss man an den Grenzen der Liste bzw. des Arrays besonders darauf achten, keine Flüchtigkeitsfehler zu machen:

```
def binary_search(sorted_values, search_for):

    mid_pos = len(sorted_values) // 2

    # Rekursiver Abbruch
    if search_for == sorted_values[mid_pos]:
        return True

    # es gibt noch mindestens 2 Zahlen
    if len(sorted_values) > 1:
        if search_for < sorted_values[mid_pos]:
            # Rekursiver Abstieg: suche im unteren Bereich weiter
            lower_half = sorted_values[0: mid_pos]
            return binary_search(lower_half, search_for)

        if search_for > sorted_values[mid_pos]:
            # Rekursiver Abstieg: suche im oberen Bereich weiter
            upper_half = sorted_values[mid_pos + 1: len(sorted_values)]
            return binary_search(upper_half, search_for)

    return False
```

Optimierter Algorithmus Die gezeigte Lösung ist nicht wirklich optimal, da zum Weitersuchen ständig Teile der ursprünglichen Daten kopiert werden. Das Ganze lässt sich unter Zuhilfenahme zweier Indexvariablen komplett ohne das potenziell aufwendige Kopieren gestalten – folgende Lösung ist sicher zu bevorzugen:

```
def binary_search_optimized(values, search_value):
    return binary_search_in_range(values, search_value, 0, len(values) - 1)


def binary_search_in_range(values, search_for, left, right):
    if right >= left:
        mid_idx = (left + right) // 2

        if search_for == values[mid_idx]:
            return True

        # Rekursiver Abstieg: suche im unteren / oberen Bereich weiter
        if search_for < values[mid_idx]:
            return binary_search_in_range(values, search_for,
                                          left, mid_idx - 1)
        else:
            return binary_search_in_range(values, search_for,
                                          mid_idx + 1, right)

    return False
```

Lösung 3b: Binärsuche iterativ (★★☆☆☆)

Wandeln Sie die rekursive Funktion in eine iterative Funktion mit abgeändertem Rückgabewert und folgendem Namen: `binary_search_iterative(values, search_for)`. Liefern Sie statt `True` bzw. `False` die Position des Suchwerts bzw. -1 für nicht gefunden.

Beispiele

Eingabe	Suchwert	Position
[1, 2, 3, 4, 5, 7, 8, 9]	5	4
[1, 2, 3, 4, 5, 7, 8, 9]	6	-1

Algorithmus Basierend auf der gerade gezeigten rekursiven Variante lässt sich die iterative Umsetzung recht einfach herleiten. Dabei nutzen wir zwei Positionsmarker $left$ und $right$ für links und rechts, die initial am Anfang und Ende (Position 0 und $length - 1$) starten. Diese beide Marker bestimmen die jeweiligen Indexgrenzen, in denen weiter gesucht werden muss. Zunächst vergleichen wir den Wert in der Mitte mit dem gesuchten. Bei Gleichheit geben wir den Index zurück. Ansonsten halbieren wir das Suchgebiet und fahren fort, bis entweder die Suche erfolgreich ist oder aber sich der linke und rechte Positionsmarker überkreuzen:

```
def binary_search_iterative(values, search_for):
    left = 0
    right = len(values) - 1

    while right >= left:

        mid_idx = (left + right) // 2

        if search_for == values[mid_idx]:
            return mid_idx

        if search_for < values[mid_idx]:
            right = mid_idx - 1
        else:
            left = mid_idx + 1

    return -1
```

Prüfung

Zum Test nutzen wir folgende Eingaben, die die korrekte Funktionsweise zeigen:

```
@pytest.mark.parametrize("sorted_values, search_for, expected",
                         [([1, 2, 3, 4, 5, 7, 8, 9], 5, True),
                          ([1, 2, 3, 4, 5, 7, 8, 9],6, False)])
def test_binary_search(sorted_values, search_for, expected):
    assert binary_search(sorted_values, search_for) == expected


@pytest.mark.parametrize("sorted_values, search_for, expected",
                         [([1, 2, 3, 4, 5, 7, 8, 9], 5, True),
                          ([1, 2, 3, 4, 5, 7, 8, 9], 6, False)])
def test_binary_search_optimized(sorted_values, search_for, expected):
    assert binary_search_optimized(sorted_values, search_for) == expected


@pytest.mark.parametrize("sorted_values, search_for, expected",
                         [([1, 2, 3, 4, 5, 7, 8, 9], 5, 4),
                          ([1, 2, 3, 4, 5, 7, 8, 9], 6, -1)])
def test_binary_search_iterative(sorted_values, search_for, expected):
    assert binary_search_iterative(sorted_values, search_for) == expected
```