Aufgabe 1: Schiebeparkplatz

Team-ID: 00960

Team: Egge Q2

Bearbeiter/-innen dieser Aufgabe:
Benedikt Biedermann, Kevin Bah, Leonie-Sophie Ilyuk, Nils Kettler, Ole Kettler, Dominik Pfeiffer, Shalina Rohdenburg, Jule Schlifke, Rebekka Schmidt

16. November 2021

Inhaltsverzeichnis

Lösungsidee

Mithilfe der Beispieldaten wird ein Parkplatz erstellt, der aus zwei Reihen besteht. In der oberen Reihe stehen die Autos, die ausgeparkt werden sollen, in der unteren befinden sich die Autos, die verschoben werden können, um den Weg freizugeben.

Zu Beginn wird jedes Auto nacheinander angeschaut und geprüft, ob dies bereits ausparken kann, oder ob eines der Schiebeautos verschoben werden muss. Je nachdem, ob das Auto von dem vorderen oder hinteren Teil des Schiebeautos blockiert wird, wird entschieden, in welche Richtung (rechts oder links) das Schiebeauto zunächst verschoben werden soll, um den geringsten Aufwand zu haben.

Bevor das Schiebeauto tatsächlich bewegt wird, wird geprüft, ob es überhaupt in beide Richtungen genug Platz dafür gibt. Für den Fall, dass es sich um ein Schiebeauto handelt, das am Rand des Parkplatzes steht, gibt es dann nur noch die Möglichkeit, es in eine Richtung zu verschieben.

Steht ein anderes Schiebeauto daneben, das den Weg blockiert, wird dies zuerst verschoben und danach folgt das Schiebeauto, das den Weg für das ausparkende Auto blockiert. Dieser Vorgang geschieht so lange, bis der Weg für das Auto freigemacht wurde. Falls neben dem Schiebeauto bereits genügend Platz ist, wird es auf kürzestem Wege verschoben, sodass das Auto direkt ausparken kann.

Für den Fall, dass während dem Schiebevorgang eine Grenze erreicht wird, wird die Schieberichtung verändert und es wird versucht, die Autos nun auf der anderen Seite zu bewegen. Eine wichtige Erkenntnis an dieser Stelle ist, dass es nur zwei Möglichkeiten gibt: Entweder werden Autos nach links verschoben oder nach rechts.

Sobald ein Auto frei ist, entsteht eine Ausgabe, die angibt, welche Autos wie weit und in welche Richtung verschoben werden müssen, um den Weg für das ausparkende Auto freizugeben.

Umsetzung

Modellierung des Parkplatzes

Um einen Parkplatz den vorgegebenen Beispieldaten entsprechend zu modellieren, wird jedes auszuparkende Auto einer zweidimensionalen Liste *parkinglot* hinzugefügt. Jede Parklücke enthält den Buchstaben des Autos, das dort parkt und einem weiteren Wert, der angibt, ob das Auto blockiert wird und wenn ja, von welchem Schiebeauto es blockiert wird.

```
[['A', 0], ['B', 0], ['C', 'H'], ['D', 'H'], ['E', 0], ['F', 'I'], ['G', 'I']]

| A | B | C | D | E | F | G |
  0   0   H   H   0   I   I
```

Figure 1: Modellierung als zweidimensionale Liste und grafische Darstellung

Status-Abfrage der Autos

Mithilfe eines *for*-Loops wird jede Parklücke einzeln angeschaut und geprüft, ob das dort stehende Auto bereits ausparken kann oder nicht. Falls ja, wird direkt ausgegeben, dass das Auto frei ist und der Loop geht weiter mit dem nächsten Auto. Wenn dies von einem Schiebeauto blockiert wird, folgt eine Überprüfung, ob es sich um ein Randauto oder ein Auto in der Mitte handelt. Daraus kann darauf geschlossen werden, ob die linke oder die rechte Seite des Autos im Weg steht:

```
for current_car in range(0, len(parkinglot)):
    ...
    # Blockiert ein Schiebeauto die Ausfahrt?
    if parking_lot_copy[current_car][1] != 0:
        while parking_lot_copy[current_car][1] != 0:
            ...
            # Auto am rechten Rand?
            if current_car == len(parkinglot) - 1:
                ...
            # Auto am linken Rand?
            if current_car == 0:
                ...
            # Auto mittig?
            if 0 < current_car < len(parkinglot) - 1:
                if parking_lot_copy[current_car + 1][1] == parking_lot_copy[current_car][1]:
                    # Wir sind auf der linken Seite des Schiebeautos
                    ...
                elif parking_lot_copy[current_car - 1][1] == parking_lot_copy[current_car][1]:
                    # Wir sind auf der rechten Seite des Schiebeautos
                    ...
    else:
        # Kein Auto blockiert die Ausfahrt
        print(f"{parking_lot_copy[current_car][0]}: ")
```

Bei den Autos in der Mitte des Parkplatzes wird dies mithilfe einer weiteren *if/elif*-Abfrage herausgefunden. Als Fixpunkt der Schiebeautos wird immer die linke Seite festgelegt. Von diesem Punkt aus werden die Autos entweder nach rechts oder links verschoben.

Grenzen und blockierende Autos

Bevor das Auto tatsächlich verschoben wird, muss eine Funktion aufgerufen werden, die mithilfe einer *if/else*-Abfrage prüft, ob der geplante Zug durch eine Grenze behindert wird. Falls keine Grenze auftaucht, wird zudem überprüft, ob ein anderes Auto in die Quere kommt. Die abgebildete Funktion gibt schließlich zwei *bool*-Werte aus, die dazu genutzt werden, auf jede Situation passend zu reagieren:

```
def is_crash_left(parking_lot_copy, moving_car):
    if moving_car - 1 >= 0:
        if parking_lot_copy[moving_car - 1][1] != 0:
            crash = True
            blocking_car = True
        else:
            crash = False
            blocking_car = False
        return crash, blocking_car
    else:
        crash = True
        blocking_car = False
        return crash, blocking_car
```

Autos verschieben

Für den Fall, dass es weder einen Zusammenstoß mit der Grenze noch mit einem anderen Auto gibt, wird das Schiebeauto direkt verschoben. Da all dies innerhalb eines while-Loops geschieht, wird das aktuelle Schiebeauto bei jedem Durchlauf nur um eine Stelle verschoben, bis der Loop zu False wird und die schlussendliche Print-Ausgabe gemacht wird.

Taucht jedoch im Verlauf eine Grenze auf, die verhindert, dass die Schiebeautos weiter verschoben werden können, wird automatisch die Schieberichtung *direction* geändert. Das Schiebeauto wird anschließend so weit wie nötig verschoben, um das Auto ausparken zu lassen.

Sonderfall „mehrere Schiebeautos nacheinander verschieben“: Lösung mit Rekursion

Da es vorkommen kann, dass ein anderes Schiebeauto das Weiterschieben behindert, wurde eine Rekursion in die Funktion eingebaut. Diese prüft jedes im Weg stehende Auto und verschiebt dann das letzte dieser Reihe, sodass alle anderen nachrücken können. Auch dies geschieht, bis das ausparkende Auto frei ist.

Die gelb markierte Stelle zeigt, wo die Funktion sich selbst aufruft: Wenn es keinen *crash* und kein *blocking_car* gibt, wird hier in der oberen Reihe zwei Positionen (ein Schiebeauto) nach links weitergegangen, die rekursive Funktion ausgeführt und im Anschluss das verschieben vorgenommen.

```
def move_left_recursion(parking_lot_copy, moving_car, current_car, cars):
    crash, blocking_car = is_crash_left(parking_lot_copy, moving_car)
    if not crash:
        direction = "links"
        if not cars.get(parking_lot_copy[moving_car][1]):
            cars[parking_lot_copy[moving_car][1]] = 0
        cars[parking_lot_copy[moving_car][1]] += 1
        moving_car = move_car_one_left(parking_lot_copy, moving_car)
    elif crash and not blocking_car:
        direction = "rechts"
    else:
        direction = "links"
        moving_car -= 2
        move_left_recursion(parking_lot_copy, moving_car, current_car, cars)
        moving_car += 2
        if not cars.get(parking_lot_copy[moving_car][1]):
            cars[parking_lot_copy[moving_car][1]] = 0
        cars[parking_lot_copy[moving_car][1]] += 1
        moving_car = move_car_one_left(parking_lot_copy, moving_car)
return moving_car, direction
```

Ausgabe

Während des gesamten Vorgangs werden die Autos, die verschoben werden müssen, zusammen mit der Anzahl an Schritten, die sie verschoben werden müssen, in einem Dictionary gespeichert. Zusätzlich wird die Schieberichtung zurückgegeben. Um die Ausgabe richtig zu formatieren, wird ein for-Loop verwendet, mit dem aus den gespeicherten Verschiebungen eine Liste erstellt wird. Diese Liste enthält nun alle nötigen Daten pro Schiebeauto, das bewegt werden muss: Buchstabe, Schrittanzahl und Richtung. Anschließend wird die Liste sortiert, ggf. rückwärts gedreht und mithilfe der Methode *.join(list)* zu einem String zusammengefasst. Daraus folgt eine formatierte Print-Ausgabe.

```
def print_result(parking_lot_copy, current_car, cars, direction):
    solution = []
    for key, value in cars.items():
        move = f"{key} {value} {direction}"
        solution.append(move)
    solution.sort()
    if direction == "rechts":
        solution.reverse()
    result = ", ".join(solution)
    print(f"{parking_lot_copy[current_car][0]}: {result}")
```

Beispiele

parkplatz0.txt

7 Felder mit 2 Schiebeautos, die erste Beispielaufgabe. Problematisch sind die Autos *F*, da hier zum ersten Mal mehr als ein Auto verschoben werden musste, und *G*, weil die „Wand” als Grenze leicht zu einem *IndexError* führen kann.

```
| A | B | C | D | E | F | G |
          H   H       I   I
A:
B:
C: H 1 rechts
D: H 1 links
E:
F: H 1 links, I 2 links
G: I 1 links
```

parkplatz1.txt

Doppelt so viele (14) Felder mit doppelt so vielen (4) Schiebeautos wie im ersten Beispiel. Allerdings dennoch ähnliche Schwierigkeit, es müssen nicht mehr als zwei Schritte für jedes Feld gemacht werden.

```
| A | B | C | D | E | F | G | H | I | J | K | L | M | N |
      O   O   P   P       Q   Q           R   R
A:
B: P 1 rechts, O 1 rechts
C: O 1 links
D: P 1 rechts
E: O 1 links, P 1 links
F:
G: Q 1 rechts
H: Q 1 links
I:
J:
K: R 1 rechts
L: R 1 links
M:
N:
```

parkplatz2.txt

14 Felder mit 5 Schiebeautos, wobei hier zum ersten Mal drei Schiebeautos direkt nebeneinander stehen. Wenn man weniger Autos bewegen wollte, könnte für *F* auch „O 1 links, P 2 rechts" gewählt werden – diese Lösung benötigt aber ebenso drei Schritte. Ein neuer Fall mit fünf Schritten ist *M*: Hier müssen, wegen der äußeren Begrenzung neben *S* auch alle nebeneinanderstehenden Autos (*R*, *Q*, *P*) bewegt werden:

```
| A | B | C | D | E | F | G | H | I | J | K | L | M | N |
        O   O       P   P   Q   Q   R   R       S   S
A:
B:
C: O 1 rechts
D: O 1 links
E:
F: R 1 rechts, Q 1 rechts, P 1 rechts
G: P 1 links
H: R 1 rechts, Q 1 rechts
I: P 1 links, Q 1 links
J: R 1 rechts
K: P 1 links, Q 1 links, R 1 links
L:
M: P 1 links, Q 1 links, R 1 links, S 2 links
N: S 1 links
```

parkplatz3.txt

14 Felder mit 5 Schiebeautos und somit auf den ersten Blick identisch mit dem vorangegangenen Beispiel. Hier müssen für *M* mehrere Autos jeweils zwei Schritte bewegt werden.

```
| A | B | C | D | E | F | G | H | I | J | K | L | M | N |
      O   O       P   P           Q   Q   R   R   S   S
A:
B: O 1 rechts
C: O 1 links
D:
E: P 1 rechts
F: P 1 links
G:
H:
I: R 1 rechts, Q 1 rechts
J: Q 1 links
K: R 1 rechts
L: Q 1 links, R 1 links
M: Q 2 links, R 2 links, S 2 links
N: Q 1 links, R 1 links, S 1 links
```

parkplatz4.txt

16 Felder mit 5 Schiebeautos. Die Besonderheit ist hier, dass nun am die Autos am linken Rand des Parkplatzes nebeneinander stehen.

```
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P |
  Q   Q   R   R           S   S           T   T       U   U
A: R 1 rechts, Q 1 rechts
B: R 2 rechts, Q 2 rechts
C: R 1 rechts
D: R 1 links
E:
F:
G: S 1 rechts
H: S 1 links
I:
J:
K: T 1 rechts
L: T 1 links
M:
N: U 1 rechts
O: U 1 links
P:
```

parkplatz5.txt

15 Felder mit 4 Schiebeautos. Hier konnten wir keine neue Schwierigkeit erkennen.

```
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
          P   P   Q   Q           R   R           S   S
A:
B:
C: Q 1 rechts, P 1 rechts
D: P 1 links
E: Q 1 rechts
F: P 1 links, Q 1 links
G:
H:
I: R 1 rechts
J: R 1 links
K:
L:
M: S 1 rechts
N: S 1 links
O:
```

Bonusparkplatz

An dieser Stelle enden die Beispielaufgaben, jedoch fiel uns beim Testen mit extremen Bedingungen wie der unteren auf, dass unser Algorithmus bei *D, F, H, J, L, N* einen Bug zu haben scheint, da *P* ignoriert wird. Es erscheint auf Anhieb unlogisch, dass „links” als Schieberichtung gewählt wird. Es scheint auch nur jeder zweite Buchstabe betroffen zu sein. Leider fehlte uns am Ende die Zeit, um den Code auch für derartige Eingaben zu optimieren.

```
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
  P   P   Q   Q   R   R   S   S   T   T   U   U   V   V
A: V 1 rechts, U 1 rechts, T 1 rechts, S 1 rechts, R 1 rechts, Q 1 rechts, P 1 rechts
B: V 1 rechts, U 2 rechts, T 2 rechts, S 2 rechts, R 2 rechts, Q 2 rechts, P 2 rechts
C: V 1 rechts, U 1 rechts, T 1 rechts, S 1 rechts, R 1 rechts, Q 1 rechts
D: Q 1 links
E: V 1 rechts, U 1 rechts, T 1 rechts, S 1 rechts, R 1 rechts
F: Q 1 links, R 1 links
G: V 1 rechts, U 1 rechts, T 1 rechts, S 1 rechts
H: Q 1 links, R 1 links, S 1 links
I: V 1 rechts, U 1 rechts, T 1 rechts
J: Q 1 links, R 1 links, S 1 links, T 1 links
K: V 1 rechts, U 1 rechts
L: Q 1 links, R 1 links, S 1 links, T 1 links, U 1 links
M: V 1 rechts
N: Q 1 links, R 1 links, S 1 links, T 1 links, U 1 links, V 1 links
O:
```

Quellcode

```
from pathlib import Path
from copy import deepcopy

def read_input(filename='parkplatz5.txt'):
    """ Beispieldatei einlesen
    Die Zeilen in Integer und List umwandeln und Zeilenumbrüche mit .strip() entfernen.
    Default ist das Aufgabenbeispiel parkplatz0.txt.
    """
    file = Path('sampledata', filename)
    with open(file, 'r') as file_in:
        parked_cars = file_in.readline().split()
        moving_cars_total = file_in.readline().strip()
        moving_cars = [line.strip() for line in file_in.readlines()]
    return parked_cars, moving_cars


def make_parkinglot(parked_cars, moving_cars):
    """ Erstellt aus parked_cars und alphabet die Liste parkinglot mit allen Autos, die ausgeparkt
    werden sollen und dem dazugehörigen Wert, der angibt, ob die Stelle blockiert wird oder frei ist
    """
    alphabet = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q",
                "R", "S", "T", "U", "V", "W", "X", "Y", "Z"]
    parkinglot = []
    end = parked_cars[1]

    occupiedlot = {}

    for element in moving_cars:
        letter, number = element.split()
        number = int(number)
        occupiedlot[letter] = number
        occupiedlot[letter.lower()] = number + 1

    count = 0
    for letter in alphabet:
        spot = [letter, 0]
        for key, value in occupiedlot.items():
            if value == count:
                spot.append(key.upper())
                spot.remove(0)
        parkinglot.append(spot)
        if letter == end:
            break
        count += 1

    print(f"Layout Parkplatz: {parkinglot}\n")
    return parkinglot


def move_cars(parkinglot):
    """ Geht durch den gesamten Parkplatz und entscheidet, was getan werden muss, damit die Autos
    ausparken können.
    """
```

```
    moving_car = None
    for current_car in range(0, len(parkinglot)):
        parking_lot_copy = deepcopy(parkinglot)
        cars = {}
        iteration_count = 0
        direction = None
        # Blockiert ein Schiebeauto die Ausfahrt?
        if parking_lot_copy[current_car][1] != 0:
            while parking_lot_copy[current_car][1] != 0:
                iteration_count += 1
                if iteration_count == 1:
                    # Fall 1: Wir befinden uns im ersten Durchlauf, nur ein Auto muss verschoben
                              werden

                    # Auto am rechten Rand?
                    if current_car == len(parkinglot) - 1:
                        moving_car = current_car - 1
                        moving_car, direction = move_left_recursion(parking_lot_copy, moving_car,
                                                                    current_car, cars)
                        continue

                    # Auto am linken Rand?
                    if current_car == 0:
                        moving_car = current_car
                        moving_car, direction = move_right_recursion(parking_lot_copy, moving_car,
                                                                     current_car, cars)
                        continue

                    # Auto mittig?
                    if 0 < current_car < len(parkinglot) - 1:
                        if parking_lot_copy[current_car + 1][1] == parking_lot_copy[current_car][1]:
                            # Wir sind auf der linken Seite des Schiebeautos
                            moving_car = current_car
                            moving_car, direction = move_right_recursion(parking_lot_copy,
                                                                         moving_car, current_car, cars)
                            continue

                       elif parking_lot_copy[current_car - 1][1] == parking_lot_copy[current_car[1]:
                            # Wir sind auf der rechten Seite des Schiebeautos
                            moving_car = current_car - 1
                            moving_car, direction = move_left_recursion(parking_lot_copy,
                                                                        moving_car, current_car, cars)
                            continue
                else:
                    # Fall 2: Mehrere Autos blockieren und müssen verschoben werden
                    if direction == "links":
                        moving_car, direction = move_left_recursion(parking_lot_copy, moving_car,
                                                                    current_car, cars)
                        continue
                    elif direction == "rechts":
                        moving_car, direction = move_right_recursion(parking_lot_copy, moving_car,
                                                                     current_car, cars)
                        continue

            print_result(parking_lot_copy, current_car, cars, direction)
        else:
            # Fall 0: Kein Auto blockiert die Ausfahrt
            print(f"{parking_lot_copy[current_car][0]}: ")
```

```
def move_left_recursion(parking_lot_copy, moving_car, current_car, cars):
    """ Prüft, inwiefern Autos nach links verschoben werden können und verschiebt sie, sobald
    genügend Platz vorhanden ist.
    """
    crash, blocking_car = is_crash_left(parking_lot_copy, moving_car)
    if not crash:
        direction = "links"
        if not cars.get(parking_lot_copy[moving_car][1]):
            cars[parking_lot_copy[moving_car][1]] = 0
        cars[parking_lot_copy[moving_car][1]] += 1
        moving_car = move_car_one_left(parking_lot_copy, moving_car)
    elif crash and not blocking_car:
        direction = "rechts"
    else:
        direction = "links"
        moving_car -= 2
        move_left_recursion(parking_lot_copy, moving_car, current_car, cars)
        moving_car += 2
        if not cars.get(parking_lot_copy[moving_car][1]):
            cars[parking_lot_copy[moving_car][1]] = 0
        cars[parking_lot_copy[moving_car][1]] += 1
        moving_car = move_car_one_left(parking_lot_copy, moving_car)

    return moving_car, direction

def move_right_recursion(parking_lot_copy, moving_car, current_car, cars):
    """ Prüft, inwiefern Autos nach rechts verschoben werden können und verschiebt sie, sobald
    genügend Platz vorhanden ist.
    """
    crash, blocking_car = is_crash_right(parking_lot_copy, moving_car)
    if not crash:
        direction = "rechts"
        if not cars.get(parking_lot_copy[moving_car][1]):
            cars[parking_lot_copy[moving_car][1]] = 0
        cars[parking_lot_copy[moving_car][1]] += 1
        moving_car = move_car_one_right(parking_lot_copy, moving_car)
    elif crash and not blocking_car:
        direction = "links"
    else:
        direction = "rechts"
        moving_car += 2
        move_right_recursion(parking_lot_copy, moving_car, current_car, cars)
        moving_car -= 2
        if not cars.get(parking_lot_copy[moving_car][1]):
            cars[parking_lot_copy[moving_car][1]] = 0
        cars[parking_lot_copy[moving_car][1]] += 1
        moving_car = move_car_one_right(parking_lot_copy, moving_car)

    return moving_car, direction

def move_car_one_left(parking_lot_copy, moving_car):
    """ Verschiebt das Auto um eine Stelle nach links.
    """
    parking_lot_copy[moving_car - 1][1] = parking_lot_copy[moving_car][1]
    parking_lot_copy[moving_car + 1][1] = 0
    moving_car -= 1
    return moving_car
```

```python
def move_car_one_right(parking_lot_copy, moving_car):
    """ Verschiebt das Auto um eine Stelle nach rechts.
    """
    parking_lot_copy[moving_car + 2][1] = parking_lot_copy[moving_car][1]
    parking_lot_copy[moving_car][1] = 0
    moving_car += 1
    return moving_car

def is_crash_left(parking_lot_copy, moving_car):
    """ Prüft, ob das Auto auf der linken Seite durch eine Grenze blockiert wird oder nicht.
    Falls daneben keine Grenze liegt, wird überprüft, ob es durch ein anderes Auto blockiert wird.
    """
    if moving_car - 1 >= 0:
        if parking_lot_copy[moving_car - 1][1] != 0:
            crash = True
            blocking_car = True
        else:
            crash = False
            blocking_car = False
        return crash, blocking_car
    else:
        crash = True
        blocking_car = False
        return crash, blocking_car

def is_crash_right(parking_lot_copy, moving_car):
    """ Prüft, ob das Auto auf der rechten Seite durch eine Grenze blockiert wird oder nicht.
    Falls daneben keine Grenze liegt, wird überprüft, ob es durch ein anderes Auto blockiert wird.
    """
    if moving_car + 2 <= len(parkinglot) - 1:
        if parking_lot_copy[moving_car + 2][1] != 0:
            crash = True
            blocking_car = True
        else:
            crash = False
            blocking_car = False
        return crash, blocking_car
    else:
        crash = True
        blocking_car = False
        return crash, blocking_car

def print_result(parking_lot_copy, current_car, cars, direction):
    """ Macht das dictionary zu einer Liste und formatiert die Ausgabe so, dass die Autos in der
    richtigen Reihenfolge genannt werden.
    """
    solution = []
    for key, value in cars.items():
        move = f"{key} {value} {direction}"
        solution.append(move)
    solution.sort()
    if direction == "rechts":
        solution.reverse()
    result = ", ".join(solution)
    print(f"{parking_lot_copy[current_car][0]}: {result}")
```