Aufgabenblatt 4

Allgemeine Informationen zum Aufgabenblatt:

- Die Abgabe erfolgt in TUWEL. Bitte laden Sie Ihre Python-Datei bis spätestens **Donnerstag, 01.05. 16:00 Uhr** in TUWEL hoch.
- Beachten Sie bitte folgende Punkte
  - Die Programme müssen syntaktisch korrekt sein. Achten Sie außerdem darauf, dass die Beispiele aus der Angabe zu keinen Abstürzen führen.
  - Wenn Fehler auftreten ziehen wir abhängig von der Schwere der Fehler Punkte ab. Bei entsprechend kleinen Fehlern können auch gar keine Punkte abgezogen werden. Geben Sie daher die bestmögliche eigene Lösung ab, auch wenn diese nicht vollständig richtig ist.
  - Ihre Programme sollen demonstrieren, dass Sie die Themen des Aufgabenblattes beherrschen. Verwenden sie daher keine "Abkürzungen" und beschränken Sie sich auf die Konstrukte die in der Vorlesung vorgestellt wurden.
  - Um das Testen einzelner Aufgaben zu erleichtern sollen alle Aufgaben als entsprechende Funktionen implementiert werden. Bitte achten Sie darauf, dass Sie in der Abgabedatei keinen Code außerhalb der in den jeweiligen Aufgaben definierten Funktionen erstellen. Sie finden entsprechende Kommentare in der Angabedatei.
  - Bei den meisten Angaben sind explizite Annahmen definiert. Sie können davon ausgehen, dass diese Annahmen stets eingehalten werden. D.h. Sie müssen deren Gültigkeit weder überprüfen, noch muss Ihre Implementierung für Werte welche diese Annahmen nicht erfüllen funktionieren.
  - Das Aufgabenblatt enthält 4 Aufgaben, auf welche Sie insgesamt 5 Punkte erhalten können.

In diesem Aufgabenblatt werden folgende Themen behandelt:

- Rekursion
- Kollektionen (Listen, Tupel, Mengen (Sets))

3.4.2025: Vorbedingung `len(line) > 0` für `biggest_char` in Beispiel 1 hinzugefügt.

Aufgabe 1 (Rekursion) [1 Punkt]

Ziel dieser Aufgabe ist es, den Entwurf einfacher **rekursiver Funktionen** zu üben.
Implementieren Sie die vier angegebenen Funktionen. Verwenden Sie zu deren Umsetzung keine Schleifen, globalen Variablen oder zusätzliche Funktionen. Operatoren (`+`, `-`, `*`, `%`, ...) sowie Slicing sind erlaubt. Auch die Funktionsköpfe dürfen nicht erweitert oder verändert werden.
Testen Sie Ihre Implementierung zumindest mit den angegebenen Aufrufen, indem Sie die Ergebnisse der Aufrufe in der Konsole ausgeben. Verwenden Sie dazu die Funktion `exercise1()`, d.h. geben Sie die Aufrufe und Ausgaben bitte in dieser Funktion an.

- `sum_up(begin, end)`: berechnet die Summe $(3 \cdot i - 2)$ für i im Intervall `[begin, end]` und gibt diese zurück. Z.B. berechnet `sum_up(3, 4)` die Summe $(3 \cdot 3 - 2) + (3 \cdot 4 - 2) = 17$.

  Annahme: `begin`, `end` sind ganze Zahlen (int) und `begin` $\leq$ `end`.

  Beispielaufrufe und erwartete Rückgaben:

`sum_up(3, 4)`	gibt `17` zurück
`sum_up(-3, 3)`	gibt `-14` zurück
`sum_up(2, 2)`	gibt `4` zurück
`sum_up(-4, 8)`	gibt `52` zurück

- `add_digits(expression)`: addiert die in `expression` dargestellten positiven und negativen Werte und gibt die Summe zurück. `expression` ist ein String gerader Länge in dem sich immer eines der Rechenzeichen `'+'` und `'-'` mit einer Ziffer abwechselt. Summiert all diese (positiven und negativen) Zahlen von links nach rechts auf und gibt das Ergebnis (als int) zurück. Für den leeren String `''` wird `0` zurückgegeben.

  Annahmen: `expression` ist ein String wie oben beschrieben.

  Beispielaufrufe und erwartete Rückgaben:

`add_digits('')`	gibt `0` zurück
`add_digits('+3+2+1')`	gibt `6` zurück
`add_digits('+9-3+8+7')`	gibt `21` zurück
`add_digits('-4-4+5-0')`	gibt `-3` zurück

- `divide_by_seven(begin, end)`: gibt einen neuen String zurück, welcher alle restlos durch 7 teilbaren Zahlen im Intevall `[begin, end]` enthält. Vor der ersten, nach der letzten und zwischen allen Zahlen steht jeweils ein `'|'`. Enthält das Intervall keine restlos durch 7 teilbare Zahl wird nur `'|'` zurückgegeben.

  Annahmen: `begin` und `end` sind ganze Zahlen (int) mit `begin` $\leq$ `end`.

  Beispielaufrufe und erwartete Rückgaben:

`divide_by_seven(5, 5)`	gibt `|` zurück
`divide_by_seven(1, 6)`	gibt `|` zurück
`divide_by_seven(-7, 7)`	gibt `|-7|0|7|` zurück
`divide_by_seven(2, 65)`	gibt `|7|14|21|28|35|42|49|56|63|` zurück

- `biggest_char(line)`: hängt an den Anfang von `line` das lexikographisch größte[1] Zeichen in `line` an und gibt den verlängerten String zurück.

  Annahmen: `line` ist ein String mit `len(line) > 0`.

  Beispielaufrufe und erwartete Rückgaben:

`biggest_char('a')`	gibt `aa` zurück
`biggest_char('aA')`	gibt `aaA` zurück
`biggest_char('12')`	gibt `212` zurück
`biggest_char('abcdefg')`	gibt `gabcdefg` zurück
`biggest_char('ABcDEF')`	gibt `cABcDEF` zurück
`biggest_char('You thought that through')`	gibt `uYou thought that through` zurück

[1] *Hinweis:* der lexikographische Vergleich zwischen Strings wird in Python mit `<` und `>` durchgeführt.

Aufgabe 2 (Kollektionen (1) – Sequenzen [Listen, Tupel, Strings]) [1 Punkt]

Ziel dieser Aufgabe ist es, vertrauter im Umgang mit Sequenzen (speziell Listen und Tupeln) zu werden. Dafür sollen einige einfache Funktionen auf diesen implementiert werden. Sie dürfen zur Lösung der Aufgaben alle in der Vorlesung behandelten Sprachkonstrukte verwenden, sowie die in Python verfügbaren Funktionen für Kollektionen.

- Implementieren Sie die beschriebenen Funktionen und
- testen Sie Ihre Implementierung zumindest mit den angegebenen Aufrufen.

Geben Sie alle zum Testen nötigen Aufrufe in der Funktion `exercise2()` an.

1. Eine Funktion `enclose(values)`, welche eine neue Liste zurückgibt. Diese enthält als ersten Eintrag den kleinsten und als letzten Eintrag den größten Wert in `values`. Dazwischen enthält sie *alle* Einträge von `values` in der selben Reihenfolge wie in `values`.

   Annahme: `values` ist eine Liste für die $0 <$ `len(values)` gilt. Der kleinste sowie größte Wert in `values` ist sinnvoll definiert (d.h. `values` enthält nur Werte die untereinander mit `<` und `>` vergleichbar sind; also z.B. nur Zahlen, nur Zeichenketten, etc.)

   Beispielaufrufe und erwartete Ergebnisse:

   `enclose([3, 2, 1, 6, 5, 4])` gibt `[1, 3, 2, 1, 6, 5, 4, 6]` zurück

   `enclose(['a'])` gibt `['a', 'a', 'a']` zurück

   `enclose([1, 2, 0, 0, 4, 5])` gibt `[0, 1, 2, 0, 0, 4, 5, 5]` zurück

   `enclose([[], [3], [1, 1, 5]])` gibt `[[], [], [3], [1, 1, 5], [3]]` zurück

2. Eine Funktion `extract(seq)`, welche ein Tupel mit fünf Einträgen zurückgibt. Der erste und letzte Eintrag des Tupels sind der erste bzw. letzte Eintrag in `seq`. Der dritte Wert v des Tupels ist der Wert in der Mitte von `seq` (sollte `seq` eine gerade Anzahl an Einträgen besitzen, dann ist v der *rechte* der beiden mittleren Werte). Der zweite und vierte Wert im Tupel geben jeweils die Anzahl an Einträgen an, welche in `seq` zwischen dem ersten Eintrag und v bzw. zwischen v und dem letzten Eintrag liegen.

   Annahme: `seq` ist eine Sequenz mit `len(seq)` ≥ 3

   Beispielaufrufe und erwartete Ergebnisse:

   `extract([1, 2, 3, 4, 5])` gibt `(1, 1, 3, 1, 5)` zurück

   `extract((1, 2, 3, 4))` gibt `(1, 1, 3, 0, 4)` zurück

   `extract('abc')` gibt `('a', 0, 'b', 0, 'c')` zurück

   `extract(('G', [1, 2], 5, 'P', 2, 5, 'A'))` gibt `('G', 2, 'P', 2, 'A')` zurück

3. Eine Funktion `find_first_average(candidates, goal)`, welche in der Liste `candidates` die ersten drei aufeinanderfolgenden Stellen sucht, deren Durchschnitt (berechnet als Summe der Stellen dividiert durch 3) maximal um 0.00001 von dem Wert von `goal` abweicht[2]. Die Funktion gibt eine neue Liste zurück: Wird eine entsprechende Stelle in `candidates` gefunden, dann besitzt die neue Liste 4 Einträge: der erste Eintrag ist der Index an dem die drei Stellen in `candidates` gefunden wurden (also der Index des ersten der drei Werte), die restlichen Stellen sind die drei Werte aus `candidates`. Enthält `candidates` keine entsprechenden Elemente (das inkludiert den Fall, dass `candidates` überhaupt weniger als drei Einträge besitzt) wird eine Liste mit dem einzigen Eintrag `-1` zurückgegeben.

   Annahme: `candidates` ist eine Liste von Zahlen (int und float), und `goal` ist eine Zahl (int oder float).

   Beispielaufrufe und erwartete Ergebnisse:

`find_first_average([1, 1, 1, 1, 1, 1], 1)`	gibt `[0, 1, 1, 1]` zurück
`find_first_average([1, 4, 2, 1, 1.5, 1], 1.5)`	gibt `[2, 2, 1, 1.5]` zurück
`find_first_average([2, 2, 2], 2)`	gibt `[0, 2, 2, 2]` zurück
`find_first_average([4, 3, 3, 3], 3)`	gibt `[1, 3, 3, 3]` zurück
`find_first_average([1, 1, 1, 1, 1, 1], 3)`	gibt `[-1]` zurück
`find_first_average([1, 1], 1)`	gibt `[-1]` zurück

[2]Hier werden möglicherweise float-Werte verglichen. Diese sollten wegen möglichen Rundungs- und Darstellungsungenauigkeiten nicht auf Gleichheit überprüft werden.

Aufgabe 3 (Kollektionen (2)) [1 Punkt]

Genauso wie in Aufgabe 2 ist das Ziel der Aufgabe vertraut im Umgang mit Sequenzen wie Listen und Tupeln, sowie anderen Kollektionen (vor allem Mengen), zu werden. Außerdem geht es darum, die einheitliche Verwendung von Kollektionen zu üben.
Implementieren Sie die beschriebenen Funktionen und testen Sie diese zumindest mit den gegebenen Aufrufen. Verwenden Sie zum Testen die Funktion `exercise3()`. Es sind sämtliche in der Vorlesung behandelten Sprachkonzepte inklusive der in Python verfügbaren Funktionen für Kollektionen erlaubt.

1. Eine Funktion `split(values, threshold)`, welche ein Tupel mit zwei Listen zurückgibt. Die erste Liste enthält alle Elemente aus `values` die kleiner als `threshold` sind. Die zweite Liste enthält die restlichen Werte. Die Reihenfolge dieser Einträge entspricht in beiden Listen der Reihenfolge in `values`, wenn `values` eine Sequenz ist, und ist sonst unbestimmt. Beide Listen enthalten zusätzlich als ersten Eintrag die Anzahl der jeweiligen Werte in `values`.

   Annahmen: `values` ist eine Kollektion und alle Einträge in `values` lassen sich mit `threshold` vergleichen (mittles `<` und `>`).

   Beispielaufrufe und erwartete Ergebnisse:

   `split([1, 7, 3, 9, 10], 5)`
   gibt `([2, 1, 3], [3, 7, 9, 10])` zurück

   `split({'ich', 'will', 'aber', 'nicht'}, 'mama')`
   gibt `([2, 'aber', 'ich'], [2, 'will', 'nicht'])` zurück

   `split((['N', 'C', 'C'], ['-'], ['1701']), ['enterprise'])`
   gibt `([3, ['N', 'C', 'C'], ['-'], ['1701']], [0])` zurück

   `split('Na Na Na', 'a')`
   gibt `([5, 'N', ' ', 'N', ' ', 'N'], [3, 'a', 'a', 'a'])` zurück

   `split([], 2025)`
   gibt `([0], [0])` zurück

2. Eine Funktion `winners(games)`: `games` ist eine Kollektion von vierstelligen Tupeln der Form `(str, int, int, str)` welche Sportergebnisse repräsentieren. Die beiden Strings stellen die Namen der Teams dar, und die jeweils danebenstehende Zahl die von diesem Team erreichten Punkte. Es gewinnt das Team mit der größeren Punkteanzahl. Die Funktion gibt eine Menge mit allen Namen von Teams zurück, die mindestens einmal gewonnen haben.

   Annahmen: `games` ist eine Kollektion von Tupeln der zuvor beschriebenen Struktur.

   Beispielaufrufe und erwartete Ergebnisse (die Reihenfolge in Mengen kann variieren):

   ```
   winners([('AUT', 0, 9, 'ESP'), ('ENG', 1, 0, 'GER'), ('GER', 7, 1, 'BRA'),
            ('ITA', 0, 0, 'ARG')])
   ```
   gibt `{'ESP', 'GER', 'ENG'}` zurück

   `winners({('A', 1, 2, 'AA'), ('BBB', 3, 2, 'BB'), ('A', 0, 0, 'C')})`
   gibt `{'AA', 'BBB'}` zurück

   `winners(set())`
   gibt `set()` zurück

3. Eine Funktion `shared_by_all(values, *collections)`: prüft für jeden Eintrag in `values` ob er in allen Kollektionen in `*collections`, welche mindestens drei Einträge haben, vorkommt. Gibt eine Menge mit allen Einträgen aus `values` zurück, welche diese Eigenschaft erfüllen. Daraus ergibt sich auch, dass die zurückgegebene Menge alle Einträge aus `values` enthält, wenn es in `*collections` gar keine Kollektion mit mindestens drei Einträgen gibt.

   Annahme: alle übergebenen Argumente sind Kollektionen, alle Einträge in `values` dürfen Werte von Mengen sein (z.B. keine Listen).

   Beispielaufrufe und erwartete Ergebnisse (die Reihenfolge in Mengen kann variieren):

   `shared_by_all({1, 2, 3, 4}, {1, 3}, {2, 3, 4}, [2, 4, 5])`
   gibt `{2, 4}` zurück

   `shared_by_all('Hallo', {1, 3}, 'hello', ('l', 'o', 'p'))`
   gibt `{'o', 'l'}` zurück

   `shared_by_all([(1,), (2,)], [(1,), tuple(), (2,)], ((3,), (2,), (1,)), 'na')`
   gibt `{(1,), (2,)}` zurück

   `shared_by_all('Singapore', 'ja', {1, 2}, [[], []])`
   gibt `{'e', 'i', 'n', 'S', 'g', 'o', 'p', 'a', 'r'}` zurück

4. Eine Funktion `top_donors(donations)`: `donations` ist eine Kollektion von zweistelligen Tupeln, welche an erster Stelle jeweils einen Namen (als String) und an zweiter Stelle wiederum ein Tupel enthalten. Dieses Tupel enthält eine unbestimmte Anzahl an positiven Zahlen (welche Spendensummen darstellen sollen).

   Die Funktion erzeugt für jedes Tupel in `donations`, welches mehr als zwei Spenden enthält oder bei dem die Summe der Spenden 500 übersteigt, ein neues, dreistelliges Tupel. Dieses enthält an erster Stelle den Namen des ursprünglichen Eintrages, an zweiter Stelle die Summe der Spenden des Eintrags, und an dritter Stelle die Anzahl an Spenden. Die Funktion gibt eine Menge mit den erzeugten Tupeln zurück.

   Anmerkung 1: Jedes Tupel in `donations` wird getrennt betrachtet. Scheint derselbe Name in mehreren Tupeln auf, müssen diese Tupel nicht kombiniert werden (siehe 1. Testfall).

   Anmerkung 2: Sollte das selbe Ergebnistupel öfter erzeugt werden, scheint es in der Ergebnis*menge* natürlich nur einmal auf (siehe 2. Testfall) – dies widerspricht nicht Anmerkung 1.

   Annahme: `donations` ist eine Kollektion mit dem zuvor beschriebenen Inhalt.

   Beispielaufrufe und erwartete Ergebnisse (die Reihenfolge in Mengen kann variieren):

   ```
   top_donors([('Alice', (200, 300)), ('Bob', (10, 10, 10)), ('Eve', (20, 50)),
               ('Alice', (1000,)), ('Bob', (300, 300))])
   ```
   gibt `{('Bob', 600, 2), ('Alice', 1000, 1), ('Bob', 30, 3)}` zurück

   ```
   top_donors([('GPA', (100, 100, 100)), ('GPA', (100, 100, 100)),
               ('GPA', (100, 100, 101)), ('GPA', (150, 50, 101))])
   ```
   gibt `{('GPA', 300, 3), ('GPA', 301, 3)}` zurück

   `top_donors({('A', (100, 100)), ('B', (100, ))})`
   gibt `set()` zurück

Aufgabe 4 (Geduldiges Parteienverkehr Anstellen) [2 Punkte]

Ziel dieser Aufgabe ist es, das Navigieren in und Arbeiten mit verschachtelten Listen zu üben. Dazu verwenden wir verschachtelte Listen, um Warteschlangen (auf einem Amt, vor Kassen, ...) abzubilden. Jeder Eintrag einer Liste stellt dabei eine wartende Person dar, die wir durch eine Zahl repräsentieren. Die Zahl beschreibt die Ungeduld der Person (größere Werte bedeuten, die Person ist ungeduldiger). Sie sollen nun eine etwas größere Anzahl von (jeweils recht kurzen) Funktionen zum Manipulieren dieser Warteschlangen implementieren.
Die Warteschlangen werden als Liste von Listen gespeichert, wobei jede verschachtelte Liste eine Warteschlange darstellt. Die links abgebildete Liste

```
[[1, 4, 7, 3],
 [2, 4],
 [0, 8, 11]  ]
```

```
Schalter 0: 01 04 07 03
Schalter 1: 02 04
Schalter 2: 00 08 11
```

stellt drei Schlangen dar. In der ersten stehen vier Personen, wobei die erste am geduldigsten ist, und die dritte am ungeduldigsten. In der zweiten Schlange stehen zwei – relativ – geduldige Personen, während von den drei Personen in der letzten Schlange die erste Person sehr geduldig ist, die anderen beiden aber die ungeduldigsten Personen in allen Schlangen sind.
Um Ihnen beim Entwickeln zu helfen, ist in der Datei `blatt4.py` bereits eine Funktion vorgegeben:

- `visualize_queues(queues)` gibt den Inhalt der in `queues` dargestellten Warteschlangen auf der Konsole aus. Dabei wird jeder Eintrag als zweistellige Zahl dargestellt. Am Beginn jeder Zeile wird die Nummer der Schlange ausgegeben. Als Beispiel finden Sie die Ausgabe für die weiter oben besprochene Liste ebendort dargestellt.

  Annahmen: Jede Liste (sowohl `queues` als jede verschachtelte Liste) kann leer sein, $0 \leq x \leq 99$ gilt für alle Zahlen x in `queues`.

Implementieren Sie die unten definierten Funktionen. Testen Sie jede Funktion mit passenden Aufrufen, zumindest aber mit den angegeben Aufrufen. Verwenden Sie zum Testen die Funktion `exercise4()`. Abweichend von den bisherigen Beispielen finden Sie die Beispielaufrufe und erwarteten Ergebnisse diesmal nicht direkt bei der Beschreibung jeder Funktion, sondern gesammelt nach der Beschreibung aller Funktionen.
Allgemeine Annahme: `queues` ist immer eine Liste von Listen von nicht negativen ganzen Zahlen.

1. `person_arrives(queues, qid, impatience)`: (*Eine neue Person stellt sich am Ende einer Schlange an.*) fügt den Wert `impatience` als neuen Eintrag am Ende der Liste an Index `qid` in `queues` hinzu. Die Funktion hat keine (explizite) Rückgabe.

   Annahmen: `qid` ist ein gültiger Index von `queues` und `impatience` ist ein int.

2. `time_passes(queues)`: (*Die Zeit vergeht, und die Leute werden ungeduldiger.*) erhöht den Wert jeder Zahl in `queues` um `1`. Die Funktion hat keine (explizite) Rückgabe.

3. `serve_snacks(queues, qid)`: (*allen Wartenden in Warteschlange* ***`qid`*** *werden Snacks serviert – das hebt die Stimmung*) verringert alle Werte der Zeile mit Index `qid` in `queues` um 10, jedoch nicht auf weniger als 0. Die Funktion hat keine (explizite) Rückgabe.

   Annahmen: `qid` ist ein gültiger Index von `queues`.

4. `split_most_impatient_queue(queues)`: (*Um Ausschreitungen zu vermeiden wird ein zusätzlicher Schalter aufgemacht. Dafür wird die ungeduldigste Warteschlange halbiert.*) berechnet für jede Warteschlange (Liste) in `queues` die Summe der Werte, und sucht jene Warteschlange mit der größten Summe heraus. Ist dies nicht eindeutig, wird aus den Listen mit der größten Summe jene am kleinsten Index in `queues` genommen. Enthält diese Liste mehr als einen Eintrag wird die Liste geteilt: die erste Hälfte (bei ungeraden Längen ist diese um ein Element kürzer als die zweite Hälfte) bleibt am ursprünglichen Index in `queues`, die zweite Hälfte wird als neuer Eintrag an das Ende von `queues` hinzugefügt. Um die Implementierung einfach zu halten geschieht die Entscheidung, ob eine Liste geteilt wird, immer auf der Liste mit maximaler Ungeduld (bzw. jener mit kleinstem Index bei Gleichstand). Ist diese zu kurz wird *keine* andere Liste geteilt. Die Funktion hat keine (explizite) Rückgabe.

   Annahmen: `len(queues)` ≥ 1

5. `customer_finished(queues, qid)`: (*In einer der Warteschlangen ist die Person vorne am Schalter fertig.*) falls in `queues` die Liste an Index `qid` nicht leer ist, wird das erste Element aus der Liste entfernt und von der Funktion zurückgegeben. Ist die Liste leer bleibt sie unverändert und es wird `-1` zurückgegeben.

   Annahme: `qid` ist ein gültiger Index von `queues`.

6. `cut_lines(queues, max_length)`: (*Der Brandschutz gibt uns leider eine maximale Anzahl an erlaubten Wartenden vor, wir schicken die Leute an den Enden der Warteschlangen nach Hause.*) kürzt alle Warteschlangen (Listen) in `queues` auf maximal `max_length` Einträge ein, indem alle Listen, die mehr als `max_length` Einträge haben, nach `max_length` vielen Einträgen "abgeschnitten" werden. Die Funktion gibt die Anzahl an weggeschickten Personen zurück. (Ein Aufteilen auf evtl. kürzere Warteschlangen findet *nicht* statt.)

   Annahmen: `max_length` ≥ 0 ist ein int.

7. `throws_tantrum(queues, position)`: (*Eine Person hat einen lautstarken Wutanfall. Das senkt deren Stimmung und die der Personen um sie herum.*) erhöht den Wert des durch `position` gekennzeichneten Eintrags in `queues` um 5, und den aller direkt angrenzten Wartenden um 2. "Direkt angrenzende" Wartende sind jene, welche entweder in der selben Warteschlange direkt vor oder nach der Person stehen, sowie die beiden Personen die in der Schlange mit der um 1 kleineren oder größeren Nummer an der selben Stelle stehen wie die Person. *Achtung:* keine dieser vier Einträge muss tatsächlich existieren.

   `position` gibt die Position der wütenden Person an: der erste Eintrag in `position` ist der Index der Warteschlange in `queues`, der zweite Eintrag die Stelle in dieser Warteschlange.

   Annahmen: `position` ist ein Tupel und beschreibt einen existierenden Eintrag in `queues`.

8. `long_distance_chat(queues, pos, from_qid, to_qid)`: (*Zwei Personen in verschiedenen Warteschlangen unterhalten sich lautstark miteinander – über die Köpfe der anderen Personen hinweg, die das nicht lustig finden.*) erhöht in `queues` in allen Zeilen mit einem Index zwischen `from_qid` und `to_qid` (beides exklusive) den Wert an Stelle `pos` (sofern dieser existiert) um 2. Die Funktion gibt zurück in wie vielen Zeilen der Wert erhöht wurde.

   Annahmen: `from_qid` und `to_qid` sind int mit $0 \leq$ `from_qid` $<$ `to_qid` $<$ `len(queues)`, und der Index `pos` existiert sowohl in `queues[from_qid]` als auch in `queues[to_qid]` (aber nicht notwendigerweise in allen Zeilen dazwischen).

Beispielaufrufe und erwartete Ergebnisse[3]:

1. Für `q1 = [[4, 3, 9, 12], [], [3, 6, 12, 24]]`:
   Nach `person_arrives(q1, 0, 2)` erzeugt `print(q1)` die Ausgabe:
   `[[4, 3, 9, 12, 2], [], [3, 6, 12, 24]]`
   Nach *anschließendem* `person_arrives(q1, 2, 48)` erzeugt `print(q1)` die Ausgabe:
   `[[4, 3, 9, 12, 2], [], [3, 6, 12, 24, 48]]`
   Nach *anschließendem* `person_arrives(q1, 1, 1)` erzeugt `print(q1)` die Ausgabe:
   `[[4, 3, 9, 12, 2], [1], [3, 6, 12, 24, 48]]`
2. Für `q2 = [[4, 3, 9, 12], [], [3, 6, 12, 24]]` und `q3 = [[]]`:
   Nach `time_passes(q2)` erzeugt `print(q2)` die Ausgabe:
   `[[], [1, 1, 3, 4, 51], [26, 13, 6], [], [2]]`
   Nach *anschließendem* `time_passes(q2)` erzeugt `print(q2)` die Ausgabe:
   `[[], [2, 2, 4, 5, 52], [27, 14, 7], [], [3]]`
   Nach `time_passes(q3)` erzeugt `print(q3)` die Ausgabe
   `[[]]`
3. Für `q4 = [[14], [1, 10, 11, 20], [], [5, 15, 25]]`:
   Nach `serve_snacks(q4, 0)` erzeugt `print(q4)` die Ausgabe:
   `[[4], [1, 10, 11, 20], [], [5, 15, 25]]`
   Nach *anschließendem* `serve_snacks(q4, 1)` erzeugt `print(q4)` die Ausgabe:
   `[[4], [0, 0, 1, 10], [], [5, 15, 25]]`
   Nach *anschließendem* `serve_snacks(q4, 1)` erzeugt `print(q4)` die Ausgabe:
   `[[4], [0, 0, 0, 0], [], [5, 15, 25]]`
   Nach *anschließendem* `serve_snacks(q4, 1)` erzeugt `print(q4)` die Ausgabe:
   `[[4], [0, 0, 0, 0], [], [5, 15, 25]]`
   Nach *anschließendem* `serve_snacks(q4, 2)` erzeugt `print(q4)` die Ausgabe:
   `[[4], [0, 0, 0, 0], [], [5, 15, 25]]`
   Nach *anschließendem* `serve_snacks(q4, 3)` erzeugt `print(q4)` die Ausgabe:
   `[[4], [0, 0, 0, 0], [], [0, 5, 15]]`
   Nach *anschließendem* `serve_snacks(q4, 3)` erzeugt `print(q4)` die Ausgabe:
   `[[4], [0, 0, 0, 0], [], [0, 0, 5]]`

[3] Aus Platzgründen geben wir hier die "normale" Ausgabe der verschachtelten Listen an, nicht die übersichtlichere Variante mittels `visualize_queues`. Sie finden in der Datei `blatt4.py` aber die Aufrufe von `visualize_queues` mit den hier abgebildeten Ergebnissen, um sich die Warteschlangen zeilenweise ausgeben lassen zu können.

4. Für `q5 = [[2, 7, 3], [10, 5], [8, 7], [2, 2, 2, 2]]` und `q6 = [[], [0, 0, 0]]`:
   Nach `split_most_impatient_queue(q5)` erzeugt `print(q5)` die Ausgabe:
   `[[2, 7, 3], [10], [8, 7], [2, 2, 2, 2], [5]]`
   Nach *anschließendem* `split_most_impatient_queue(q5)` erzeugt `print(q5)` die Ausgabe:
   `[[2, 7, 3], [10], [8], [2, 2, 2, 2], [5], [7]]`
   Nach *anschließendem* `split_most_impatient_queue(q5)` erzeugt `print(q5)` die Ausgabe:
   `[[2], [10], [8], [2, 2, 2, 2], [5], [7], [7, 3]]`
   Nach *anschließendem* `split_most_impatient_queue(q5)` erzeugt `print(q5)` die Ausgabe:
   `[[2], [10], [8], [2, 2, 2, 2], [5], [7], [7, 3]]`[4]
   Nach `split_most_impatient_queue(q6)` erzeugt `print(q6)` die Ausgabe:
   `[[], [0, 0, 0]]`[5]

5. Für `q7 = [[1, 2, 3], [4], [5, 6]]`:
   `customer_finished(q7, 0))` gibt `1` zurück und `print(q7)` erzeugt die Ausgabe:
   `[[2, 3], [4], [5, 6]]`
   Jeweils *anschließend* an den vorherigen Aufruf ausgeführt erhalten wir:
   `customer_finished(q7, 1))` gibt `4` zurück und `print(q7)` erzeugt die Ausgabe:
   `[[2, 3], [], [5, 6]]`
   `customer_finished(q7, 1))` gibt `-1` zurück und `print(q7)` erzeugt die Ausgabe:
   `[[2, 3], [], [5, 6]]`
   `customer_finished(q7, 2))` gibt `5` zurück und `print(q7)` erzeugt die Ausgabe:
   `[[2, 3], [], [6]]`
   `customer_finished(q7, 0))` gibt `2` zurück und `print(q7)` erzeugt die Ausgabe:
   `[[3], [], [6]]`

6. Für `q8 = [[1, 2, 3, 4, 5], [0], [], [10, 11, 12]]`:
   `cut_lines(q8, 3)` gibt `2` zurück und `print(q8)` erzeugt die Ausgabe:
   `[[1, 2, 3], [0], [], [10, 11, 12]]`
   Jeweils *anschließend* an den vorherigen Aufruf ausgeführt erhalten wir:
   `cut_lines(q8, 3)` gibt `0` zurück und `print(q8)` erzeugt die Ausgabe:
   `[[1, 2, 3], [0], [], [10, 11, 12]]`
   `cut_lines(q8, 2)` gibt `2` zurück und `print(q8)` erzeugt die Ausgabe:
   `[[1, 2], [0], [], [10, 11]]`
   `cut_lines(q8, 0)` gibt `5` zurück und `print(q8)` erzeugt die Ausgabe:
   `[[], [], [], []]`

[4] Die höchste Ungeduld ist 10, diese kommt zuerst in der List `[10]` vor, welche nicht mehr geteilt wird.
[5] Die Summen in beiden Listen ergibt 0, die erste solche Liste ist die leere Liste, welche nicht geteilt wird.

7. Für `[[1, 1, 1], [1, 1, 1], [1, 1, 1], [1, 1]]`:

   Nach `throws_tantrum(q9, (1, 1))` erzeugt `print(q9)` die Ausgabe:

   `[[1, 3, 1], [3, 6, 3], [1, 3, 1], [1, 1]]`

   Nach *anschließendem* `throws_tantrum(q9, (2, 2))` erzeugt `print(q9)` die Ausgabe:

   `[[1, 3, 1], [3, 6, 5], [1, 5, 6], [1, 1]]`

   Nach *anschließendem* `throws_tantrum(q9, (0, 0))` erzeugt `print(q9)` die Ausgabe:

   `[[6, 5, 1], [5, 6, 5], [1, 5, 6], [1, 1]]`

8. Für
   `q10 = [[1, 1, 1, 1], [1, 1], [1, 1, 1], [1, 1], [1, 1, 1, 1], [1, 1, 1, 1]]`:

   `long_distance_chat(q10, 1, 1, 5)` gibt `3` zurück und `print(q10)` erzeugt die Ausgabe:

   `[[1, 1, 1, 1], [1, 1], [1, 3, 1], [1, 3], [1, 3, 1, 1], [1, 1, 1, 1]]`

   Jeweils *anschließend* an den vorherigen Aufruf ausgeführt erhalten wir:

   `long_distance_chat(q10, 2, 0, 4)` gibt `1` zurück und `print(q10)` erzeugt die Ausgabe:

   `[[1, 1, 1, 1], [1, 1], [1, 3, 3], [1, 3], [1, 3, 1, 1], [1, 1, 1, 1]]`

   `long_distance_chat(q10, 3, 0, 4)` gibt `0` zurück und `print(q10)` erzeugt die Ausgabe:

   `[[1, 1, 1, 1], [1, 1], [1, 3, 3], [1, 3], [1, 3, 1, 1], [1, 1, 1, 1]]`