Assignment Guidelines:

- This assignment covers material in Module 4.
- Submission details:
  o Solutions to these questions must be placed in files a04q1.py, a04q2.py, a04q3.py, and a04q4.py, respectively, and must be completed using Python 3.
  o Download the interface file from the course Web page to ensure that all function names are spelled correctly and each function has the correct number and order of parameters.
  o All solutions must be submitted to MarkUs. No solutions will be accepted through email, even if you are having issues with MarkUs.
  o Verify using MarkUs and your basic test results that your files were properly submitted and are readable on MarkUs.
  o For full style marks, your program must follow the Python section of the CS116 Style Guide.
  o Be sure to review the Academic Integrity policy on the Assignments page.
  o Helper functions need design recipe elements but not examples and tests.
- Download the testing module from the course web page. Include import check in each solution file.
  o When a function returns a floating point value, you must use check.within for your testing. Unless told otherwise, you may use a tolerance of 0.00001 in your tests.
  o Test data for all questions will always meet the stated assumptions for consumed values.
- Restrictions:
  o Do not import any modules other than math and check.
  o Do not use Python constructs from later modules (e.g. loops). Do not use any other Python functions not discussed in class or explicitly allowed elsewhere. See the allowable functions post on Piazza. You are always allowed to define your own helper functions, as long as they meet the assignment restrictions.
  o While you may use global constants in your solutions, do not use global variables for anything other than testing.
  o Read each question carefully for additional restrictions.
  o The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.
1. Calculating your assignment grade.

Write a Python function `assignment_mark` that consumes a list of assignment grades (called `grades`), and returns a floating point value between 0.0 and 100.0, inclusive, for the overall assignment mark. Individual assignment grades are either an integer value between 0 and 100, inclusive, or the string 'x'.

In calculating the assignment grade, note the following:

- The 'x' entries are ignored, as this indicates that the student was ill and the assignment is not to be considered when calculating the assignment mark.
- The final assignment is worth double, so treat it as two assignment grades in calculations.
- A student cannot be excused from the last assignment, so the last entry in the list will never be 'x'.
- The lowest assignment grade (not counting the last assignment or 'x' grades) is dropped. In the case of ties, just one of these is dropped.
- While we will have 9 assignments this term, grades may be longer. You can assume that there will be at least three non-excluded assignments, including the last one.
- Do not perform any rounding on the calculated assignment grade, but remember to use `check.within` for your testing.

The function will return the average of all assignments, other than the excluded and minimum grade, while counting the final assignment as two assignments. The list `grades` is not changed by the function. For example,

```python
assignment_mark([100, 82, 'x', 75, 45, 'x', 80]) => 83.4
```

The returned value is the average of 100, 82, 75, 80, 80, as 45 is dropped as the lowest grade, 'x' values are ignored, and the final assignment is counted as two assignments.

2. Redacting text

Your solution to this question must use recursion. It cannot use any abstract list functions.

For the purposes of this question, we will consider a word to be a nonempty string consisting only of lowercase alphabetic characters.

Write a function `redact_text`, that consumes two lists of words, `original` and `banned`, and returns the number of times words in `banned` occur in `original`. In addition, the list `original` is mutated so that each occurrence of a word in `original` is replaced with the string "<redact>". Note that words in `banned` may occur multiple times in `original`, and the total number of times is the returned value. The list `banned` is not changed by `redact_text`.

For example, suppose

```python
speech = ["cs", "is", "fun", "homework", "is", "sometimes", "fun", "exams", "are", "not", "fun"].
```
Then `redact_text` (speech, ["fun", "is", "not"])
=> 6 and `updates` speech to
["cs", "<redact>", "<redact>", "homework", "<redact>", "sometimes",
"<redact>", "exams", "are", "<redact>", "<redact>"]).

3. It's a Knizia! (No, you don't really need to know this reference.)

*Your solution to this question must use abstract list functions. Do not use any recursion.*

In some board games (for example, *Ingenius* and *Tigris and Euphrates*), players earn points in several categories, and their score is determined to be their minimum point value in any category. For example, if a player was gathering points in five categories, and received: 10, 3, 8, 12, 9, their score would be 3.

```haskell
## A Player is a list of length >=2, where
## * the first entry in Player is of type Str
## * the remaining entries in Player are of type Nat.

In this question, you will write a function `winners`, which consumes `competitors`, a list of Player values. Each Player contains a player's name, followed by their point values in a number of categories. The function `winners` will determine the highest score among players in `competitors`, and will return the list of names of all players with that highest score. The `competitors` list is not changed by `winners`.

For example,

```
winners(["A", 5, 9, 6, 8, 9],  # "A" has score 5
        ["X", 6, 7, 6, 7, 6],  # "X" has score 6
        ["D", 9, 11, 8, 6, 9],  # "D" has score 6
        ["C", 12, 0, 0, 5, 9]])  # "C" has score 0
=> ['X', 'D']  # Highest score is 6
```

Note the following:
- There will be at least one player in `competitors`.
- No two players have the same name.
- Each player will have points in the same number of categories, and there will be at least one category of points.
- Points will always be natural numbers.
- The names in the returned list should be in the same relative order as in `competitors`. 
4. Playing (not scoring) a card game called Take5 (or 6 Nimmt):

The "board" in the game Take5 consists of 4 rows of cards. Each row is in increasing order and contains between 1 and 5 cards. An example is shown to the left. Players take turn placing new cards on the board, one at a time. New cards can only be placed at the rightmost end of a row. In the process of placing cards, a player may need to remove all the cards in a row, and will be assigned points based on the values in that row. The goal of the game is to have the lowest score after all rounds have been played. (Points are actually based on the number of "bull horns" shown on the collected cards, but we will ignore that component of the game.)

To determine where a card is to be placed, you need to examine the rightmost card in each row (70,19,8,93 in the board on the left). A card cannot be placed next to a bigger card, and must be placed next to the card its value is closest to.

For example,

- 17 must be placed in the third row, next to 8;
- 22 must be placed in the second row, next to 19;
- 89 must be placed in the first row, next to 70.

There are three possibilities when placing a new card:

1. The card can be placed, using the described guidelines, in a row with fewer than 5 cards. It becomes the new rightmost card in that row.
2. If the row already has 5 cards, the new card cannot be added. Instead, the new card must actually replace the entire row, and the player takes all the cards in that row. Consider, for example, placing 99. Based on the numbers, it must go in the fourth row. However, since that row already has 5 cards, the player will remove the cards there (13,30,50,88,93) and then 99 will become the only card in the fourth row.
3. If the new card is smaller than all the rightmost cards, then the player must remove a row. For simplicity, we will assume that the player must choose the row which has the minimum value in its rightmost position, and then the new card will become the only card in that row. (In the real game, the player can choose any row, but usually chooses the one with the fewest "bull horns", but we will use a simpler rule.) For example, suppose we are trying to place 6. It is smaller than all the rightmost cards, so the player must take the third row (containing 8), since 8 is the smallest rightmost value, and then 6 becomes the only card in that row.

Consider the following new data definitions:

```plaintext
## A Card is a Nat between 1 and 104, inclusive.
## A Row is a (listof Card) containing between 1 and 5
##   values, in strictly increasing order.
## A Board is a (listof Row) of length 4.
## No Card appears more than once in a Board.
```

So, for example, the Take 5 board in our image is represented as in my_board:

```plaintext
my_board = [[67,70], [9,18,19], [8], [13,30,50,88,93]]
```
Write a Python function `turn_take5`, that consumes `b`, of type `Board`, and `c`, of type `Card`, where `c` is different from each of the cards already in `b`, and mutates `b` to reflect the playing of `c` (as described previously). In addition, the function returns the list of cards taken off the board during the turn:

- if `c` is played on `b` without the player removing any cards (as in possibility 1) then `[]` is returned;
- if `c` should be played on a row which already contains five cards (as in possibility 2), then a row containing only `c` replaces that row, and the function returns the contents of the replaced row;
- if `c` is smaller than the rightmost card in each row of `b` (as in possibility 3), then a row containing only `c` replaces the row with the smallest rightmost card, and the function returns the contents of the replaced row.

For example, consider the following sequence of calls (in which the value of `my_board` is updated by each successive call, as indicated.

- `turn_take5 (my_board, 17) => [], and mutates my_board to [[67, 70], [9, 18, 19], [8, 17], [13, 30, 50, 88, 93]]`
- `turn_take5 (my_board, 22) => [], and mutates my_board to [[67, 70], [9, 18, 19, 22], [8, 17], [13, 30, 50, 88, 93]]`
- `turn_take5 (my_board, 99) => [13, 30, 50, 88, 93], and mutates my_board to [[67, 70], [9, 18, 19, 22], [8, 17], [99]]`
- `turn_take5 (my_board, 5) => [8, 17], and mutates my_board to [[67, 70], [9, 18, 19, 22], [5], [99]]`