NATIONAL UNIVERSITY OF SINGAPORE
Department of Computer Science, School of Computing
IT5001—Software Development Fundamentals
Academic Year 2023/2024, Semester 2
PROBLEM SET 4

SETS, DICTIONARIES & DEBUGGING

These exercises are optional and ungraded, and for your own practice only. Contact `yongqi@nus.edu.sg` for queries.

SIMPLE EXERCISES

Question 1. Without using IDLE, determine the output from the following print statements.

```
a = ('A', 2), ('B', 3), (1, 4)
set_a = set(a)
print('1.', set_a)
print('2.', 2 in set_a)
set_a.discard((1, 4))
print('3.', set_a)
set_a.discard((1, 4))
print('4.', set_a)
set_a.remove((1, 4))
print('5.', set_a)
b = [[1, 'A'], [(1, 2), 4]]
set_b = set([y for x in b for y in x])
print('6.', set_b)
print('7.', (1, 2) in set_b)
print('8.', set_a < set_b)
set_b.remove((1, 2))
print('9.', set_b)
set_b.add(1)
print('10.', set_b)
set_b.add(2)
print('11.', set_b)
set_c = set((i[0] for i in set_a))
print('12.', set_c)
set_d = set_b | set_c
print('13.', set_d)
print('14.', set_b & set_c)
print('15.', set_b - set_c)
print('16.', set_b ^ set_c)
set_c |= set_b
print('17.', set_b & set_c)
```

```
print('18.', set_b - set_c)
print('19.', set_b ^ set_c)
set_c.add(set_b)
print('20.', set_c)
print('21.', set_b.union([10, 20]))
print('22.', set_b | [10, 20])
```

Question 2. Without using IDLE, determine the output from the following print statements.

```
a = (('A', 2), ('B', 3), (1, 4))
dict_a = dict(a)
print('1.', dict_a)
print('2.', dict_a[2])
print('3.', dict_a.get(2))
b = [[1, 'A'], [(2, 3), 4]]
dict_b = dict(b)
print('4.', dict_b)
print('5.', dict_b[(2,3)])
for key in dict_b.keys():
    print('6.', key)
for val in dict_b.values():
    print('7.', val)
for key,val in dict_b.items():
    print('8.',key, val)
del dict_b[(2, 3)]
print('9.', dict_b)
del dict_b[2]
print('10.', dict_b)
print(tuple(dict_a.keys()))
print('11.', list(dict_a.values()))
dict_c = {1: {2: 3}, 4: 5}
print('12.', dict_c)
dict_d = dict_c.copy()
dict_d[4] = 9
dict_d[1][2] = 9
print('13.', dict_c)
dict_d.update({2: 10, 3: 20, 4: 7})
print('14.', dict_d)
print('15', dict_d.pop(100))
print('16', dict_d.popitem())
dict_c.clear()
print('17.', dict_c)
print('18.', dict_d)
del dict_c
print('19.', dict_c)
print('20.', dict_d[1][2])
```

SHORT PROGRAMMING QUESTIONS

Question 3. We are going to implement functions and structures that represent and operate on matrices.

Question 3 (i). Write a function `identity_matrix(n)` that takes in some integer `n` and returns an $n \times n$ identity matrix. Each cell in the identity matrix is a 1 if its row and column indices are the same, 0 otherwise. You may want to represent a matrix as a 2-dimensional list. The following is an example 4×4 identity matrix and its corresponding representation as a 2-dimensional list.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

```
[[1, 0, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 1, 0],
 [0, 0, 0, 1]]
```

Question 3 (ii). Write a function `print_matrix(m)` that takes in a matrix and prints the matrix tightly. Example runs follow.

```
>>> print_matrix(identity_matrix(4))
1000
0100
0010
0001
```

Question 3 (iii). Write a function `zero_matrix(r, c)` that takes in integers r and c and creates a new matrix with r rows and c columns containing all zeroes.

Question 3 (iv). Write a function `transpose(m)` that takes in a matrix m and returns its transpose: m^{T}.

Question 3 (v). Write a function `matrix_mult(a, b)` that performs matrix multiplication on two input matrices, a and b, assuming that the number of columns in a is equal to the number of rows in b.

Suppose we have two matrices, **A** and **B**.

$$\mathbf{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \qquad \mathbf{B} = \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,p} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,p} \end{pmatrix}$$

Then the result of multiplying **A** and **B** is a matrix **C** where

$$\mathbf{AB} = \mathbf{C} = \begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,p} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m,1} & c_{m,2} & \cdots & c_{m,p} \end{pmatrix}$$

$$c_{i,j} = a_{i,1}b_{1,j} + \cdots + a_{i,n}b_{n,j} = \sum_{k=1}^{n} a_{i,k}b_{k,j}$$

An example run follows.

```
>>> a = [[1, 2, 3],
         [5, 6, 7],
         [9, 10, 11],
         [13, 14, 15]]
>>> b = [[4, 3, 2, 1, 8, 1],
         [1, 2, 3, 4, 3, 1],
         [5, 6, 7, 8, 1, 2]]
>>> matrix_mult(a, b)
[[21, 25, 29, 33, 17, 9],
 [61, 69, 77, 85, 65, 25],
 [101, 113, 125, 137, 113, 41],
 [141, 157, 173, 189, 161, 57]]
```

Question 3 (vi). Write a function `minor_matrix(m, i, j)` that gives the matrix resulting from removing row i and column j from m. An example run follows.

```
>>> m = [[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9]]
>>> minor_matrix(m, 2, 1)
[[1, 3],
 [4, 6]]
```

Question 3 (vii). The following is the Laplace expansion for finding the determinant of a matrix $\mathbf{A}$:

$$\det(\mathbf{A}) = \sum_{i=0}^{n} (-1)^i a_{0,i} \det(M_{0,i})$$

where $M_{0,i}$ is the minor of matrix $\mathbf{A}$ where row 0 and column i have been removed.

Write a function `det(m)` that returns the determinant of some input matrix m. An example run follows.


```
>>> m = [[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9]]
>>> det(m)
0
```

Question 4. An anagram is a word or a phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once. For example, "nag a ram" is an anagram for "anagram" and "eleven plus two" is an anagram for "twelve plus one",

Write a function `is_anagram(word1, word2)` that returns `True` if the two words are anagrams of one another, `False` otherwise.

Question 5. Old mobile phones only have numerical keypads where every letter is associated with a number as shown in Figure 1. Number 0 is designated as the spacebar. Since writing is tedious in old mobile phones, the T9 predictive text technology system was created. It works as follows:

- Every word is composed of letters.
- Every letter can be mapped to a digit.
- Given a series of digits of keypresses, we can find the expected word.

We can represent the keypad in two different ways:

1. A list such that each element is a string of characters associated with the element's index. This gives us: `[' ', '', 'abc', 'def', 'ghi', 'jkl', 'mno', 'pqrs', 'tuv', 'wxyz']`
2. A dictionary where they keys are the characters and the values are the associated number. This gives us: `{'a': 2, 'b': 2, ..., 'z': 9, ' ': 0}`

Figure 1: Sample T9 layout

Suppose there are other alphabets and other symbols for larger numbers that are not restricted to just 10 digit symbols and 26 alphabets.

Question 5 (i). Write a function `to_dict(key_l)` which takes in the keys in the list representation and returns the dictionary representation.

Question 5 (ii). Write a function `to_list(key_d)` which takes in the keys in the dictionary representation and returns the list representation.

Question 5 (iii). Write a function `to_nums(word)` that takes in a string and returns a string representing the keypresses that need to be made to type `word`. Assume that you have both `key_l` and `key_d` initialized as in the image, and that your function has access to both. For example, `to_nums('i luv u')` would give `'4058808'`.

Question 5 (iv). Write a function `to_letters(num)` that takes in a number in string form and returns a list of all combinations of letters that can be represented by those keypresses. The combinations of letters may appear in any order.

Question 6. You are given a nonnegative integer n as the target amount of dollars, and a set C of positive integers as the coin denominations. Write a function `num_ways(n, C)` that returns the number of ways you can obtain $\$n$ from C as denominations. There are an unlimited supply of coins with values in C. For example, `num_ways(3, {8, 3, 1, 2})` should return 3 as there are three ways to obtain \$3: three \$1 coins, one \$1 coin and one \$2 coin, and one \$3 coin. Write a recursive solution. Is memoization helpful?

Question 7. You are given a string consisting of words separated by spaces. Write a function `k_most_freq(s, k)` that receives the string `s` and some number `k` that returns a list of the `k` most frequently occurring words in `s`, sorted in ascending order of frequency. If two words have the same number of occurrences, the words should be ordered lexicographically in your result. Use the `sorted` function to your advantage. An example run follows.

```
>>> k_most_freq('a dog walked in a dog park in a dog world', 3)
['in', 'a', 'dog']
```

1 LONG PROGRAMMING QUESTION

We are going to be playing around with mazes! Imagine a maze as a 2D grid of r rows and c columns. Each cell in the grid is either 0 or 1; 0 indicates a traversable cell, and 1 is a blocked cell. To walk around the maze, you can only move up, down, left or right, and not diagonally.

1.1 Random Maze Generator

Write a function `create_random_maze(r, c, p0 = 0.7)` that creates a maze consisting of r rows and c columns where each cell has a probability of p_0 of being a 0, and $1 - p_0$ of being a 1.

1.2 Traversable Neighbours

Write a function `neighbours(maze, r, c)` that takes in a maze and some cell coordinates, and returns a list of the traversable neighbours of the cell at the coordinates.

An example run follows.

```
>>> m = create_random_maze(2, 3)
>>> m
[[0, 0, 1],
 [0, 0, 1]]
>>> neighbours(m, 1, 1)
[(0, 1), (1, 0)]
```

1.3 Solvable

Write a function `solvable(m)` that takes in a maze and returns **`True`** if it is possible to traverse from the top left cell to the bottom right cell in the maze, **`False`** otherwise. Use BFS. Example runs follow.

```
>>> m = create_random_maze(2, 3)
>>> m
[[0, 0, 1],
 [0, 0, 1]]
>>> solvable(m)
False
>>> m = create_random_maze(3, 3)
>>> m
[[0, 0, 1],
 [1, 0, 1],
 [1, 0, 0]]
>>> solvable(m)
True
```

1.4 Shortest Path

Instead of returning **`True`** if the maze is solvable, visualize the shortest path from the top left cell to the bottom right cell (if there are multiple, then any one is fine). Note that BFS will find the shortest path already, so your only job is to keep track of the path and to visualize it. Example runs follow.

```
>>> m = create_random_maze(2, 3)
>>> m
[[0, 0, 1],
```

```
 [0, 0, 1]]
>>> solvable(m)
False
>>> m = create_random_maze(3, 3)
>>> m
[[0, 0, 1],
 [1, 0, 1],
 [1, 0, 0]]
>>> solvable(m)
[['+', '+',  1],
 [ 1 , '+',  1],
 [ 1 , '+', '+']]
```

SOLUTIONS

Question 1. Finding out the solutions to these is quite simple—just write the expressions in IDLE to see the results after evaluating them.

Some points to note:

1. Items to be added to a set must be hashable.
2. `my_set.remove(...)` vs `my_set.discard(...)`: one raises and error and the other doesn't if the item is not in the set.
3. set operators requires a set as the second operand

Question 2. Finding out the solutions to these is quite simple—just write the expressions in IDLE to see the results after evaluating them.

Some points to note:

1. Keys of a dictionary must be hashable.
2. Accessing a non-existing key using indexing will produce an error, but using the `get` method will not.

Question 3. Majority of these are relatively straightforward.

```
def identity_matrix(n):
    return [[1 if i == j else 0 for j in range(n)] for i in range(n)]

def print_matrix(m):
    for row in m:
        print(''.join(str(i) for i in row))

def zero_matrix(r, c):
    return [[0] * c for _ in range(r)]

def transpose(m):
    r, c = len(m), len(m[0])
    output = zero_matrix(c, r)
    for i in range(r):
        for j in range(c):
            output[j][i] = m[i][j]
    return output

def matrix_mult(a, b):
    r, c, n = len(a), len(b[0]), len(a[0])
    output = zero_matrix(r, c)
    for i in range(r):
        for j in range(c):
            for k in range(n):
```

```
                output[i][j] += a[i][k] * b[k][j]
    return output

def minor_matrix(m, i, j):
    return [row[:j] + row[j + 1:] for row in (m[:i] + m[i + 1:])]

def det(a):
    if len(a) == 1:
        return a[0][0]
    return sum((-1) ** i * a[0][i] * det(minor_matrix(a, 0, i))
               for i in range(len(a)))
```

Question 4. Maintain two dictionaries, one for `word1` and another for `word2`. Each dictionary maintains the number of occurrences of each character (ignoring spaces). The two words are anagrams if the dictionaries are equal.

```
def is_anagram(word1, word2):
    a, b = {}, {}
    for i in word1:
        a[i] = a.get(i, 0) + 1
    for i in word2:
        b[i] = b.get(i, 0) + 1
    a[' '] = b[' '] = 0
    return a == b
```

Question 5. Relatively straightforward.

```
def to_dict(key_l):
    return {c: i for i in range(len(key_l)) for c in key_l[i]}

def to_list(key_d):
    output = [''] * max(key_d.values())
    for k, v in key_d:
        output[v] += k
    return output

def to_nums(word):
    output = []
    for i in word:
        output.append(str(key_d[i]))
    return ''.join(output)

def to_letters(num):
    if not num:
        return ['']
    smaller_problem = to_letters(num[1:])
```

```
    letters = key_l[int(num[0])]
    return [c + s for s in smaller_problem for c in letters]
```

Question 6. Suppose $C = \{a, b, \dots\}$ and the target is n. Then, we can clearly create \$$n$ using 1 coin of \$$a$, 2 coins of \$$a$, 3 coins of \$$a$, ..., and the number of combinations we can get from doing so is $num\_ways(n-a, \{b, \dots\})$, $num\_ways(n-2a, \{b, \dots\})$, $num\_ways(n-3a, \{b, \dots\})$, ...

Then it is clear that the solution is as follows:

$$num\_ways(n, C) = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{if } n < 0 \text{ or } |C| = 0 \\ \sum_{i=0}^{\lfloor n/a \rfloor} num\_ways(n - ai, C - \{a\}) & \text{otherwise, for some } a \in C \end{cases}$$

Writing code for the above is trivial. Don't forget to perform memoization. Importantly, mutable data structures cannot be hashed, so ensure that we convert `C` into a tuple before adding it as input to the memo.

```
def num_ways(n, c, memo = None):
    # Create the memo
    if memo is None: memo = {}
    # Base cases
    if n == 0: return 1
    if n < 0 or not c: return 0
    # Get memoized solution
    if (n, tuple(c)) in memo:
        return memo[n, tuple(c)]
    # Get some element a
    a = next(iter(c))
    # Set obtained by removing a from c
    b = c - {a}
    # Maximum number of a you can have
    n_a = n // a
    # Result
    z = sum(num_ways(n - a * i, b, memo) for i in range(n_a + 1))
    # Memoize
    memo[n, tuple(c)] = z
    return z
```

Question 7. Self-explanatory, very similar to the lecture example.

```
def k_most_freq(s, k):
    # keep a dictionary to count occurrences
    d = {}
    # split the string into words
    words = s.split()
    # run a for-loop to count the number of occurrences
    for c in words:
        d[c] = d.get(c, 0) + 1
    # create a list containing pairs of (occ, word) so that we can
    # sort by occurrences
    ls = [(v, k) for k, v in d.items()]
    # sort the pairs (tuples are ordered lexicographically, just like strings)
    # and get the last k elements
    z = sorted(ls)[-k:]
    # remove the sorting key from each pair
    return [b for a, b in z]
```

Long Programming Question. We can use basic list comprehension to create a random maze.

```
from random import random
def create_random_maze(r, c, p0 = 0.7):
    return [[0 if random() < p0 else 1
             for _ in range(c)]
            for _ in range(r)]
```

To get the neighbours of a cell at some `r` and `c`, simply look top, down, left and right of the cell and check 1) if they exist and 2) the cell at that coordinates is 0.

```
def neighbours(maze, r, c):
    def is_valid(coordinates):
        # check if coordinates is a valid and traversable cell
        x, y = coordinates
        return 0 <= x < len(maze) and    \
               0 <= y < len(maze[x]) and \
               maze[x][y] == 0
    # all possible neighbours are top, bottom, left and right
    possible_neighbours = [(r - 1, c), (r + 1, c), (r, c - 1), (r, c + 1)]
    # remove the invalid neighbours
    return [i for i in possible_neighbours if is_valid(i)]
```

To determine if a maze is solvable, use BFS.

```
def solvable(m):
    # Trivial maze is where entry/exit is 1.
    if m[0][0] == 1 or m[-1][-1] == 1:
        return False
    # Frontier starts with the top-left cell's coordinates.
    frontier = [(0, 0)]
    # Visited is a set of cells. Great because of O(1) search.
    visited = set()
    while frontier:
        # Pop from front of frontier so that this is BFS.
        current_cell = frontier.pop(0)
        # Don't waste time.
        if current_cell in visited: continue
        # Goal found.
        if current_cell == (len(m) - 1, len(m[-1]) - 1): return True
        # Indicate as visited.
        visited.add(current_cell)
        # Find neighbours and add to frontier.
        frontier.extend(neighbours(m, *current_cell))
    # Goal unreachable.
    return False
```

Now to keep track of the path taken to arrive at the destination, we first keep track of two items during BFS: the cell itself, and the previous cell. Then, we first augment the `neighbours` function to also include the current cell so that the result is a list of tuples (x, y) where x is a valid neighbour of y.

```
def neighbours_with_prev(maze, r, c):
    """
    This function returns a list of (neighbour, coordinates)
    for each valid neighbour of coordinates.
    """
    return [(i, (r, c)) for i in neighbours(maze, r, c)]
```

We then augment BFS to make use of this facility. We assume that the 'previous' cell of $(0, 0)$ is itself. Instead of a visited set, we have a 'previous' dictionary that keeps track of 1) which cells have been visited, and 2) for each visited cell, the previous cell we came from to get to that cell.

```python
def solvable(m):
    """
    This function solves for the shortest path of the maze,
    and returns False if the maze is unsolvable.
    """
    # Trivial maze is where entry/exit is 1.
    if m[0][0] == 1 or m[-1][-1] == 1:
        return False
    # Frontier starts with the top-left cell's coordinates.
    frontier = [((0, 0), (0, 0))]
    # previous tells you which cell has been visited, and where it
    # came from to get to that cell.
    # Each pair is in the form of current: prev.
    # Again, great because of O(1) search.
    previous = {}
    while frontier:
        # Pop from front of frontier so that this is BFS.
        current_cell, previous_cell = frontier.pop(0)
        # Don't waste time.
        if current_cell in previous: continue
        # Keep track of where you came from.
        previous[current_cell] = previous_cell
        # Goal found.
        if current_cell == (len(m) - 1, len(m[-1]) - 1):
            # visualize_shortest_path draws the path
            return visualize_shortest_path(m, previous)
        # Find neighbours and add to frontier.
        frontier.extend(neighbours_with_prev(m, *current_cell))
    # Goal unreachable.
    return False
```

Finally, to draw the maze with the shortest path, we can write a function to do so:

```python
def visualize_shortest_path(maze, previous):
    # Deep copy the maze.
    maze = [row[:] for row in maze]
    # Start at the goal cell.
    current_cell = len(maze) - 1, len(maze[-1]) - 1
    maze[-1][-1] = maze[0][0] = '+'
    # Backtrack to the start.
    while current_cell != (0, 0):
        # Get the previous cell from the current cell
        r, c = previous[current_cell]
        # Mark the cell as .
        maze[r][c] = '+'
        # Set the previous cell as the new current cell.
        current_cell = r, c
    return maze
```