National University of Singapore
School of Computing
CS1010S: Programming Methodology

Extra Practice 7 Solutions

For Questions 1-3, you can choose any data structure to use for each question!

Question 1

We are given a certain amount of money and we are allowed to buy any 2 things in a supermarket. We are only allowed to spend the exact amount of money we are given (i.e. the 2 things we choose to buy must add up exactly to the amount we were given). Given a list containing the prices of all the items in the supermarket, define a function **`find_positions`** that takes in the list of prices, and the amount of money we have, and returns a tuple containing the two indexes, i and j, such that the list at indexes i and j both add up to the amount we have. Take note that $i \neq j$.

You can assume that there is only one correct answer (if any). If there is no possible answer, return 0.

Sample Tests:

```
>>> find_positions([4, 2, 1, 8, 5], 3)
(1, 2)  # 2+1=3
>>> find_positions([4, 2, 1, 8, 5], 5)
(0, 2)  # 4+1=5
>>> find_positions([4, 2, 1, 8, 5], 8)
0       # must be 2 items added up
>>> find_positions([4, 2, 1, 8, 5], 12)
(0, 3)  # 4+8=12
```

Note: What is the time and space complexity of your solution? Can you do better?
Solution:

```
def find_positions(lst, amt):
    for i in range(len(lst)):
        for j in range(i+1, len(lst)):
            if lst[i] + lst[j] == amt:
                return (i, j)
    return 0

# Assuming n = len(lst),
# Time: O(n**2) because we are iterating through the list
# for (n-1) + (n-2) + ... + 2 + 1 = O(n**2) times.
# Space: O(1) extra space since no slicing or new list/tuple is needed.

# O(n**2) seems to be a naive implementation, and there might exist an
# actual faster algorithm to solve this.
```

```
# Can we do better? Yes, use dictionaries! You will study this more
# in CS2040 but for now I'll give you just to let you know.

def find_positions(lst, amt):
    d = {} # store (lst[idx], idx) pairs inside dictionary

    for i in range(len(lst)): # O(n)
        comp = amt - lst[i] # we need comp to pair up with lst[i]
        if comp in d: # for dictionary, in operation is O(1)
            return tuple(sorted([i, d[comp]]))
        d[lst[i]] = i

    return 0

# Assuming n = len(lst),
# Time: O(n) time because we are only through lst once.
# Space: O(n), as a tradeoff, the dictionary has to come to be and
# can contain at most n elements.
```

Question 2

In the supermarket, there are n rows of m items in each row. Each item has a price on it. Assume that you are at the supermarket with your girlfiend (or boyfriend) who has a habit of buying everything you walk past. Given that you are at the start of the supermarket (row 1 column 1) and want to exit the supermarket located at row n column m, you wish to move in a path such that you can minimize the total amount that you need to spend. Define a function, **`cheapest_path`** that takes in a matrix and returns the least amount of money you can possibly spend. (Note: You can only move downwards or rightwards) **Sample Test:**

```
>>> matrix = [[3, 3, 1],
              [2, 9, 4],
              [7, 2, 1],
              [1, 5, 2]]
>>> cheapest_path(matrix)
14
```

Solution:

```
def submatrix(mat, i, j):
    return list(map(lambda x: x[:j], mat[:i]))

def cheapest_path(mat):
    rows = len(mat)
    cols = len(mat[0])
    if rows == 1:
        return sum(mat[0])
    elif cols == 1:
        return sum(list(map(lambda x: x[0], mat)))
    return mat[-1][-1] + min(cheapest_path(submatrix(mat, rows-1, cols)),
                             cheapest_path(submatrix(mat, rows, cols-1)))
```

```
# Alternate solution, no slicing needed
def cheapest_path(mat):
    def helper(i, j):
        if i == 1:
            return sum(mat[0])
        elif j == 1:
            return sum(list(map(lambda x: x[0], mat)))
        return mat[i-1][j-1] + min(helper(i, j-1), helper(i-1, j))

    return helper(len(mat), len(mat[0]))
```

Note: How does our code change if we now want to start from the bottom left corner, move either upwards or rightwards until we reach the top right corner?
Solution:
To do that, we can simply rotate the matrix or transpose it, then apply the same function again. Recall the tutorial and recitation problems that discussed about matrix transposing.

Question 3

Two words are considered to be *isomorphic* if each letter in one word can be mapped over to another letter. For instance, the words "aba" and "dad" are isomorphic where the letter "a" can be mapped over to "d", and the letter "b" can be mapped over to "a". In contrast, "deeds" and "poopp" are not considered isomorphic because more than one letter ("d" and "s") maps to the same letter "p". "deeds" and "poops" are isomorphic because "d" $\rightarrow$ "p", "e" $\rightarrow$ "o", "s" $\rightarrow$ "s".

Define a function **`check`** that takes in two words as inputs and returns **`True`** if both words are isomorphic to each other, and **`False`** otherwise.

Sample Test:

```
>>> check("aba", "dad")
True
>>> check("aba", "dae")
False
>>> check("aba", "bab")
True
>>> check("deeds", "poops")
True
>>> check("deeds", "poopp")
False
```

Solution:

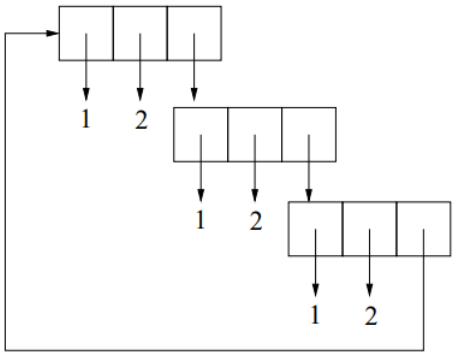
```
def check(s1, s2):
    s1_to_s2 = {}
    s2_to_s1 = {}
    for i in range(len(s1)): # assume len(s1) == len(s2)
        if s1[i] not in s1_to_s2:
            s1_to_s2[s1[i]] = s2[i]
```

```
        elif s1_to_s2[s1[i]] != s2[i]:
            return False
        if s2[i] not in s2_to_s1:
            s2_to_s1[s2[i]] = s1[i]
        elif s2_to_s1[s2[i]] != s1[i]:
            return False
    return True
```

Question 4 (WARNING: Challenging!)

(CS1010X AY16/17 Special Term I Finals, abridged)

In this problem, we will work with recursive lists of a special form where the last element points to the next list and the last list in the series points back to the first list in the series. Each list is also called a *link* (of the chain). This is illustrated here:



(a) The function **`chain(seq, n)`** creates a chain of n links by replicating a sequence **`seq`** as follows:

```
>>> chain((1, 2), 1)
[1, 2, [...]]
>>> chain((1, 2), 3) # Illustrated above
[1, 2, [1, 2, [1, 2, [...]]]]
>>> chain((4,), 4)
[4, [4, [4, [4, [...]]]]]
```

Give a possible implementation for **`chain(seq, n)`**.
Solution:

```
def chain(seq,n):
    ans = list(seq)
    ans.append(None) # placeholder
    link = list(ans)
    current = ans
    for i in range(n-1):
        new_link = list(link)
        current[-1] = new_link
        current = new_link
    current[-1] = ans
    return ans
```

(b) Given a chain it would certainly be helpful to be able to count the number of links in the chain. For example:

```
>>> count_links(chain((1, 2), 1))
1
>>> count_links(chain([3, 2], 2))
2
>>> count_links(chain((4,), 4))
4
```

Give a possible implementation for **`count_links`**.
Solution:

```
def count_links(chain):
    count, current = 1, chain[-1]
    while not current is chain:
        count, current = count + 1, current[-1]
    return count
```

(c) A regular chain has repeated links. It is possible for a fault to happen and one of the links might become "faulty". A faulty link has one element that is different from all the other links. Implement the function **`find_fault`** that will return the different element. You can assume that the chain has at least 3 links and there's only one faulty element.

```
>>> a = chain((1, 2), 4)
>>> a[2][2][0] = 5
>>> a
[1, 2, [1, 2, [5, 2, [1, 2, [...]]]]]
>>> find_fault(a)
5
```

Solution:

```
def find_fault(chain):
    depth = count_links(chain)
    found = {}
    for i in range(depth):
        if tuple(chain[:-1]) not in found:
            found[tuple(chain[:-1])] = 0
        found[tuple(chain[:-1])] += 1
        chain = chain[-1]
    keys = list(found.items())
    keys.sort(key=lambda x: x[1], reverse=True)
    keys = list(map(lambda x:x[0],keys))
    for i in range(len(keys)):
        if keys[0][i] != keys[1][i]:
            return keys[1][i]
```

Solution compiled by Russell Saerang.