The University of Melbourne
Code Masters 2017
Solutions

Matt Farrugia
matt.farrugia@unimelb.edu.au

March 23, 2017

Question 1: Number Finder

Read the numbers into a grid. Store 'marks' in another grid (because numbers may overlap). Scan through the first grid looking for matches*, and mark them in the second grid. Count how many grid squares have not been marked.

```
def count_remaining(filename):
    f = open(filename)
    n = int(f.readline())
    grid = []
    bools = []
    for i in range(n):
        grid.append(f.readline().split())
        bools.append([False] * n)

    m = int(f.readline())
    for i in range(m):
        integer = list(f.readline().strip())
        l = len(integer)
        for i in range(n):
            for j in range(n):
                if integer == grid[i][j:j+l]:
                    bools[i][j:j+l] = [True] * l
    count = 0
    for i in range(n):
        for j in range(n):
            if not bools[i][j]:
                count += 1
    return count
```

*There's lots of room for improvement over a basic, brute force scan-and-match. Not necessary for these input sizes. (Good idea to try a simple solution and see if it works before spending lots more time making a better solution.)

Question 2: Balanced Brackets

Take each line one character at a time. If it's an opening bracket, push it on top of a stack. If it's a closing bracket, match it with the top bracket on the stack. If it's not a match, or the stack is empty, the line is unbalanced. If there are brackets left unmatched at the end, it's unbalanced.

```
def unbalanced_lines(filename):
    f = open(filename)

    count = 0
    for line in f:
        balanced = True
        stack = []
        for c in line:
            if c in "{[(<":
                stack.append(c)
            elif c in "}])>":
                if len(stack) < 1 or stack.pop() != opening_version(c):
                    balanced = False
                    break
        if not balanced or len(stack) > 0:
            count += 1
    return count

def opening_version(close):
    if close == "}": return "{"
    if close == "]": return "["
    if close == ")": return "("
    if close == ">": return "<"
```

Question 3: Word Morphs

Scan each line, comparing consecutive words. To detect valid word morphs, first make sure the words are the same length, then count in how many positions the characters in each word differ (if it's < 1 or > 1, not a valid morph). A valid line must have no invalid morphs.

```
def neighbours(word1, word2):
    if len(word1) != len(word2):
        return False

    changed_a_letter = False
    for i in range(len(word1)):
        if word1[i] != word2[i]:
            if changed_a_letter:
                return False
            else:
                changed_a_letter = True
    return changed_a_letter
```

```
def count_morphs(filename):
    f = open(filename)

    count = 0
    for line in f:
        words = line.split()
        prev_word = words[0]
        valid_morph = True
        for curr_word in words[1:]:
            if not neighbours(prev_word, curr_word):
                valid_morph = False
                break
            prev_word = curr_word
        if valid_morph:
            count += 1

    return count
```

Question 4: Word Morphs Harder

Begin with the the start word. Find all valid word morphs in `words.txt` for this start word. Find all valid word morphs for *those* words. And continue, until you reach the end word. Count how many words you had to use to get here. Multiply by the length of the words to get the number of characters needed. Since we are exploring the network of words in a 'breadth first' manner, the first path to the end word will always be the shortest possible.
Make sure to only explore words that have not been explored already, otherwise we will explore forever looking for words if no path exists. We can do this with a hash map, and we can also use it to store path fragments so that we can reconstruct the path later.
How do we find valid word morphs for a given word?

- generate all possible valid morphs by changing one letter at a time, then looking them up in the list of valid words (in a dictionary/hash set)?
- scan through the list of valid words one at a time, checking if they are valid by counting different letters (like in question 3)?

How many possible valid morphs are there? If L is the length of our start word, 25 choices to change each letter: 25^L. But there are only 26^L possible words of length L, and far fewer in `words.txt`. May be better just to scan the list once.

```
def search(start, goal, d):

    prev = {start: start}
    queue = collections.deque()
    queue.append(start)

    while queue:
        u = queue.popleft()
        if u == goal:
            break
        for v in d:
            if v not in prev and neighbours(u, v):
                queue.append(v)
                prev[v] = u

    if goal not in prev:
        return 0, []

    words_on_path = 1
    w = goal
    path = [goal]
    while prev[w] != w:
        w = prev[w]
        path.append(w)
        words_on_path += 1

    path.reverse()
    return words_on_path, path
```

We can also split `words.txt` up by word length, so that we only scan through words of the same length (and don't waste time on longer/shorter words that will never be valid word morphs).

```
f = open("words.txt")
len_dict = {}
for line in f:
    word = line.strip()
    if len(word) in len_dict:
        len_dict[len(word)].append(word)
    else:
        len_dict[len(word)] = [word]
```

Question 5: Number Square

Simulation approach

Intuitive approach of just counting the squares as you spiral outwards, counting upwards. Step along each edge, until you come back to the top. Step up to the next 'layer', and then repeat. Stop when you reach the given position, and print the count.

```
def simulate(x, y):

    if (0, 0) == (x, y): return 1

    number = 2
    i, j = (0, 1)
    if (i, j) == (x, y): return number

    layer = 1

    while True:
        while i < layer:
            i += 1
            number += 1
            if (i, j) == (x, y): return number
        while j > -layer:
            j -= 1
            number += 1
            if (i, j) == (x, y): return number
        while i > -layer:
            i -= 1
            number += 1
            if (i, j) == (x, y): return number
        while j < layer:
            j += 1
            number += 1
            if (i, j) == (x, y): return number
        while i < 0:
            i += 1
            number += 1
            if (i, j) == (x, y): return number

        j += 1
        if (i, j) == (x, y): return number

        layer += 1
```

Only problem? For part 5, the answer is over 2,000,000,000,000 (2 trillion) so this will take over 2 trillion steps. That's few minutes in C++. In Python, a few days?

Direct calculation approach

A (much) faster approach is to perform the calculation directly somehow, like you might calculate the answer on paper. The key idea is to realise that the grid is made up of concentric square layers. Find out which layer the point is in, calculate the number of grid squares of all the inner layers (area) and then calculate how far along the outermost layer the point is.

```
def number(x, y):

    # what layer are we in?
    layer = max(abs(x), abs(y))

    # how many squares inside this layer?
    core_area = (2 * layer - 1) ** 2

    # how far are we along the layer?
    number = core_area + 1
    if y == layer and x >= 0:
        return number + x
    number += layer
    if x == layer:
        return number + (layer - y)
    number += 2 * layer
    if y == -layer:
        return number + (layer - x)
    number += 2 * layer
    if x == -layer:
        return number + (layer + y)
    number += 2 * layer
    return number + (layer + x)
```