Unimelb Code Masters 2015
Solutions Lecture

9 April 2015

1 Square Roots

The Newton-Raphson method allows for successive approximations to a function's value. In particular, if the first guess at the $\sqrt{k}$ is x, then a better guess is $\frac{1}{2}(x + k/x)$. This equation can be applied over and over, improving the approximation each time. More formally, the i'th guess at $\sqrt{k}$ can be written as

$$x_i = \begin{cases} k, & \text{if } i = 1 \\ \frac{1}{2}(x_{i-1} + k/x_{i-1}), & \text{otherwise} \end{cases}$$

In the following questions, give your answer to 2 decimal places.

1.1 Question 1

What is x_2 if $k = 1000$?

Absolute classic programming pattern: the accumulator-loop.

1. Initialise an accumulator variable
2. Loop over something
3. If some condition
4. update the accumulator
5. Return/Print the accumulator value

```
x = k
for ii in range(2, i+1):
    if True:
        x = x/2.0 + k/x/2.0
print("{0:2}".format(x))
```

2 Water pools

Imagine a 2D slice of a landscape, shown on its side. The # characters indicate rocky ground. The vertical dimension shows the height of the rocks.

Suppose that it rains heavily and the rocky landscape fills up with water like so (the '-' character indicates water):

```
         #
  #------#
  ##--#-##
 ###--####
 #########
 243113235
```

The height of the rocks in the landscape dictates how high the water can fill before it starts spilling over the sides.

Your task is to write a program which will read in a description of a rocky landscape as described above and print out the maximum number of units of water that can be retained in the landscape. The answer will be a non-negative integer.

2.1 Brute Force

- Treat rocks as a grid of height max(height) and width number of rocks
- Loop over each row keeping a toggle "in dip" or "not in dip" (Another accumulator loop.)
- What is the running time proportional to?

```
total = 0
for row in range(highest, 0, -1):
    counting = False
    for col in range(len(data)):
        if data[col] >= row and not counting:
            counting = True; count = -1
        elif data[col] >= row and counting:
            total += count ; count = -1

        if counting:
            count += 1
```

2.2 Iterative fill approach

- Find the lowest point and the left and right "walls"
- Count the water that is in the rectangle (left-right)*(min(left-right)-low),
- Fill it in with rocks and go again until no "bowl"
- If a low point only has one side, throw away the entire missing side

```
        #                       #                       #                       #
 #------#                 #------#                 #------#                 ########
 ##--#-##                 #####-##                 ########                 ########
###--####       →        #########       →        #########       →        #########
#########                #########                #########                #########
243113235                243333235                243333235                243333235
012345678                012345678                012345678                *12345678
total = 0  +(5-2-1)*(2)  total = 4  +(6-4-1)*(1)  total = 5  +(8-1-1)*(1)  total = 11
```

2.3 Bernie's cool stack approach...

3 Langton's Ant

Imagine a simple simulation of an ant on a rectangular two-dimensional grid of black and white square cells.

The ant walks about the grid one step at a time, obeying the following two rules:

1. If the ant is standing on a white cell, it changes the cell to black, turns right ninety degrees and takes one step forward.
2. If the ant is standing on a black cell, it changes the cell to white, turns left ninety degrees and takes one step forward.

If the ant reaches any of the edges of the grid it moves to the cell on the opposite side. In other words its movement wraps around at the edges of the grid.

The following web page contains a visualisation of the ant behaviour:

`http://students.informatics.unimelb.edu.au/~bjpope/foc/mywork/langton.py`

- No trick here, just code it!
- Be careful of your boundary conditions.

```
import numpy as np

'''
   Facing N = 0, E = 1,  S = 2, W = 3
'''
def ant(nrow, ncol, row, col, facing, n):

    a = np.zeros((nrow, ncol))

    for i in range(n):
        if a[row,col] == 0:
            a[row,col] = 1
            facing = (facing + 1) % 4
        else:
            a[row,col] = 0
            facing = (facing - 1) if facing > 0 else 3

        if facing == 0:
            row = (row - 1) if row > 0 else (nrow - 1)
        elif facing == 2:
            row = (row + 1) if row < nrow-1 else 0
        elif facing == 1:
            col = (col + 1) if col < ncol-1 else 0
        else:
            col = (col - 1) if col > 0 else (ncol - 1)

    return( (row, col))
```

4 Melbourne Tram Network

The following five questions all relate to the data in the single `trams.txt` file which contains lines describing tram stops on tram routes in Melbourne. Each line has two numbers separated by a space. The first is a stop number and the second is a route number. For example, the first 5 lines are

```
2382 11
1440 19
2330 57
3557 30
2106 5
```

Question 1 How many different routes are there in the network?

Question 2 How many different tram stops are there in the network?

Question 3 Which tram route has the most stops?

Question 4 What is the maximum number of routes that go through any one tram stop?

Question 5 What is the maximum number of stops that are shared by any two tram routes?

- This is the perfect place for a dictionary/map/associative array as in Python/Java/C++ STL/awk/perl/...
- Keep two dictionaries
  - One keyed by route with a list of stops as the payload (Q1 & Q3)
  - One keyed by stop with a list of routes as the payload (Q2 & Q4)
- For Q5, a language with set intersection (Python, R, Matlab) works well. Simply loop over all pairs of routes and intersect their stop lists and populate another dictionary(!) keyed by size of the intersection (number of shared stops).

```
d_rr = collections.defaultdict(list)
for r1 in route_dict.keys():
    for r2 in route_dict.keys():
        if r1 != r2:
            d_rr[len(set(route_dict[r1]) & set(route_dict[r2]))] += [[r1, r2]]

m = max(d_rr.keys())
```

5 Perfect Shuffle

Ace McPoker prides herself on the ability to shuffle two decks of cards perfectly, so that the cards from the two decks strictly alternate when she is finished.

In this schematic, each card is represented by two characters, the first being one of `A23456789TJQK` representing the value of the card, and the second one of `CDHS` representing the suit of the card.

When Ace shuffles, the first card in the resulting deck is always from Deck 1, and the second from Deck 2. The decks always contain the same number of cards, so the last card is always from Deck 2.

Positions are counted from 1 being the first card in the deck, and can range from 1 to 104 in the final shuffled deck, and 1 to 52 in the starting deck.

Question 1 After her first shuffle, what is the lowest position of a Jack of Hearts (`JH`)?

Question 2 After her first shuffle, what is the lowest position of a 4 of Diamonds (`4D`)?

Question 3 After her first shuffle, Ace splits the shuffled deck exactly in half so that the first 52 cards form Deck 1, and the second 52 cards form Deck 2. If she then shuffles these decks once, what is the lowest position of a King of Diamonds in the resulting deck?

Question 4 If, beginning with the two starting decks, Ace shuffles 5 times, exactly splitting the decks into two new decks of 52 as in Question 3, what is the lowest position of a Queen of Clubs after the 5 shuffles?

Question 5 What is the minimum number of shuffles after the first that is required to get a King of Spades back into position 52?

No tricks here - code fast!

Bernie's code is quick and nice. Mine is quick and dirty.

6 Jumping up stairs

Ferdinand is a fitness instructor. One of his favourite exercises is to jump up flights of stairs.

He wonders how many different ways he can jump up a flight of N stairs if he chooses between 1, 2 and 3 stairs at each jump. He must start at the bottom of the stair case and always land exactly on the top stair with the final jump.

He works out the easy cases in his head. For example, if there is only 1 stair there is only 1 way to do it. If there are 4 stairs then there are 7 solutions, as shown below where each line shows a possible sequence of jumps that sums to 4.

```
1 1 1 1
1 1 2
1 2 1
1 3
2 1 1
2 2
3 1
```

Write a program to help Ferdinand calculate the solution for larger values of N. The input to your program is an integer number of steps between 1 and 40. The output should be the number of possible ways Ferdinand can jump up the stairs.

How many possible ways can Ferdinand jump up 5,6,10,29,35 stairs?

6.1 Brute Force

- Try all possible combinations of $\{1, 2, 3\}^N$ and see which ones sum to N.
- How many are there?

N	Combos
4	$3^4 = 81$
5	$3^5 = 243$
6	$3^6 = 729$
29	$3^{29} \approx 2^{46}$

N	Seconds
19	0.60
20	1.30
21	2.79
22	5.73
23	11.6

(This was with my "smart" brute force.)

- At this rate will take about 20 minutes for $N = 29$ and 14 hours for $N = 35$.

6.2 Dynamic Programming or Memoization

- First, write the problem as a recursive function. $M(N)$ is the number of ways he can jump up N stairs using combinations of 1,2 or 3.

$$M(N) = \begin{cases} M(N-1) + M(N-2) + M(N-3) & \text{if } N \geq 3 \\ M(N-1) + M(N-2) & \text{if } N = 2 \\ M(N-1) & \text{if } N = 1 \\ 1 & \text{if } N < 1 \text{ (tricky!)} \end{cases}$$

- If you implement this, you get same as brute force
- But look closely - $M(N)$ only depends on $M(i)$ for $i < N$
- Dynamic Programming - start at 0 and build up a solution (funny name is historical)
- Memoization - use recursion, but remember values you have already calculated