Midterm Examination

CSC 326 - Programming Languages
Fall 2011

October 27, 2011

University of Toronto
Department of Computer Science

Clearly print your name and student number below:

Name:	
Student Number:	

Grade:

Question	Mark
1	
2	
3	
4	
Total:	

Problem 1. (12 points) Below is a list of code blocks. Circle the letter beside each code block that never causes an error to occur when the code is run.

A
```
ls=["a","b","c","d","e","f",]
print ",".join([letter.upper() for letter in ls])
```

B
```
ls=["a","b","c","d","e","f",]
print ",".join(letter.upper() for letter in ls)
```

C
```
as_list = [("key1", 1), ("key2", 2), ]
print {k:v for k, v in as_list}
```

D
```
my_list = {1,2,3,4,5,6,}
print my_list[1:3]
```

E
```
matrix = [[1,0,0],
          [0,1,0],
          [0,0,1]]
print matrix[0,1]
```

F
```
def is_true(res):
    if res:
        return "True"
    return "False"

numbers = [0,1,2,3,4,5]

if is_true(False):
    print numbers
else:
    print numbers[0] / 0
```

Problem 2. (18 points) Below is a list of valid Python code blocks that never result in errors. In the box provided below each code block, write out the output of the program.

A

```
ls = [0,1,2,3,4]
print ls is ls
print ls is ls[:]
```

B

```
ls = [0,1,2,3,4]
print [ls[len(ls) - i] for i in ls[1:]]
```

C

```
ls = range(2)
print [ls.append(ls[-2] + ls[-1]) or ls[-2] for _ in range(8)]
```

Hint: `list.append` returns `None`.
Hint: if `a=[1,2,3]` then `a[-1]` returns `3`.

D

```
parts = [[list(part) for part in str(x).split(".")] for x in [10.55, 19.11]]
print parts
parts = zip(*parts)
print parts
parts = zip(*parts[0]) + zip(*parts[1])
print parts
parts = [str(int(a) + int(b)) for (a, b) in parts]
print parts
print float("".join(parts[:2]) + "." + "".join(parts[2:]))
```

Hint: if `a=[[1,2],['a','b']]` then `zip(*a)` returns `[(1, 'a'), (2, 'b')]`.
Hint: if `a="abaca"` then `a.split("a")` returns `["","b","c",""]`.

Problem 3. (15 points) If s and t are two strings then we say the longest common subsequence of s and t is some string $s_{i_0}s_{i_1}\ldots s_{i_{n-1}} = t_{j_0}t_{j_1}\ldots t_{j_{n-1}}$ such that $0 \le i_0 < i_1 < \ldots < i_{n-1} < |s|$ and $0 \le j_0 < j_1 < \ldots < j_{n-1} < |t|$, where $|s|$ and $|t|$ denotes the length of the string s and t, respectively. For example, if $s = abbacad$ and $t = abaccad$ then the longest common subsequence of s and t is $abacad$, and has length 6. The following highlights the common letters of the subsequence: $s = \underline{ab}b\underline{acad}$ and $t = \underline{abac}c\underline{ad}$.

The length of the longest common subsequence of two strings can be computed as follows:

```
def LCS(s,t,i=-1,j=-1):
    # the only subsequence in common between a string and the
    # empty string is the empty string
    if i < -len(s) or j < -len(t):
        return ""

    # if the ith character of s is the jth character of t, then
    # extend the longest common subsequence found by "shortening"
    # both s and t with s[i].
    # e.g.
    #     ababa[d]      abca[d]
    #     take:
    #         LCS(ababa, abca) + d
    if s[i] == t[j]:
        return LCS(s, t, i-1, j-1) + s[i]

    # the two characters are not equal, find the longest one by
    # "shortening" either s or t by one letter
    # e.g.
    #     ababa[d]      abca[e]
    #     take the longer of:
    #         LCS(ababad, abca)
    #         LCS(ababa, abcae)
    else:
        long_from_i = LCS(s, t, i-1, j)
        long_from_j = LCS(s, t, i, j - 1)
        if len(long_from_i) < len(long_from_j):
            return long_from_j
        else:
            return long_from_i
```

The above algorithm works backward through `s` and `t`, starting from the end (because `i` and `j` default to `-1` when not explicitly specified) and computes the longest common subsequence. When run, `LCS("abbacad", "abaccad")` will return `"abacad"`.

Unfortunately, the above algorithm is very slow, particularly because it will end up re-computing various intermediate results.

Convert the above recursive algorithm into a Python function `LCS_fast(s,t)` that uses `for` loops and a data structure of your choice, such that the data structure can represent the following table, and the algorithm fills the table out in a similar manner to below. Your function should return the longest common subsequence of two arbitrary strings `s` and `t` and should *not* repeat work.

		a	b	b	a	c	a	d
	""	""	""	""	""	""	""	""
a	""							
b	""							
a	""							
c	""							
c	""							
a	""							
d	""							

$\rightarrow \ldots \rightarrow$

		a	b	b	a	c	a	d
	""	""	""	""	""	""	""	""
a	""	"a"	"a"	"a"	"a"	"a"	"a"	"a"
b	""	"a"	"ab"					
a	""							
c	""							
c	""							
a	""							
d	""							

Answer:

Problem 4. (15 points) Given an $n \times n$ grid/matrix of live/dead (live = `1`, dead = `0`) cells, a single iteration of Conway's Game of Life applies one of the following rules to each cell of the grid (note: different cells will satisfy different rules) in order to get the next state of the grid.

1. Any live cell with fewer than two live neighbours dies, as if caused by under-population.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if by overcrowding.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

These rules can be implemented in Python as follows:

```
def iterate(board):
    """board is a list of lists in row-major form."""

    num_rows, num_cols = len(board), len(board[0])

    # duplicate the old board, but extend it with rows/cols of zeros
    # along the edges so that we don't need to do bounds checking, but
    # instead do 1-based indexing instead of 0-based indexing
    old_board = [[0] * (num_cols + 2) for _ in xrange(num_rows + 2)]
    for r in xrange(num_rows):
        for c in xrange(num_cols):
            old_board[r+1][c+1], board[r][c] = board[r][c], 0

    # count the neighbors of each cell and apply the rules
    for r in xrange(1, num_rows + 1):
        for c in xrange(1, num_cols + 1):
            num_neighbors = old_board[r-1][c] + old_board[r][c-1] + \
                            old_board[r-1][c-1] + old_board[r+1][c] + \
                            old_board[r][c+1] + old_board[r+1][c+1] + \
                            old_board[r-1][c+1] + old_board[r+1][c-1]

            if old_board[r][c]:
                board[r-1][c-1] = int(num_neighbors in (2, 3))
            else:
                board[r-1][c-1] = int(3 == num_neighbors)
        return board
```

For example, the following is three iterations of Conway's Game of Life when run on the 3×3 starting grid:

$$\begin{array}{|c|c|c|}\hline 1 & 1 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 1 & 0 \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|}\hline 1 & 1 & 0 \\ \hline 0 & 1 & 1 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|}\hline 1 & 0 & 1 \\ \hline 1 & 0 & 1 \\ \hline 0 & 1 & 0 \\ \hline \end{array}$$

Use NumPy's slicing and broadcasting features, create a function `iterate(board)` that performs one iteration of Conway's Game of Life on two dimension NumPy array. You will be penalized if you use `for` loops. Hint: count neighbor cells!

Answer:

This page is intentionally left blank as a worksheet.