CS 161 Fall 2017: Section 7 Solutions

Encoding

Suppose we encode lowercase letters into a numeric string as follows: we encode a as 1, b as 2 ... and z as 26. Given a numeric string S of length n, develop an $O(n)$ algorithm to find how many letter strings this can correspond to.

Let $T(i)$ denote the number of ways to encode the string up to and including $S[i]$. We establish our recurrence as follows:

$$T(i) = \sum \begin{cases} T(i-1) & S[i] \in 1 \dots 9 \\ T(i-2) & S[i-1:i+1] \in 10 \dots 26 \end{cases}$$

To compute this in a single forward pass using dynamic programming, we build a table for each $T(i)$ and initialize base cases for $T(0 \dots 2)$

```
def num_encodings(numeric_string):
    # Initialize dp array
    # t[i] will store how many possible encodings there are for s[:i]
    t = [0 for _ in range(len(numeric_string))]

    # base cases for 0 and 1
    t[0] = int(numeric_string[0]) > 0
    two_digit_num = int(numeric_string[:2])
    t[1] = two_digit_num > 10 and two_digit_num <= 26
    if int(numeric_string[1]) > 0:
        t[1] += t[0]

    # main dp loop
    for i in range(2, len(numeric_string)):
        if int(numeric_string[i]) > 0:
            t[i] += t[i-1]

        two_digit_num = int(numeric_string[i-1:i+1])
        if two_digit_num > 10 and two_digit_num <= 26:
            t[i] += t[i-2]

    # return value at last index
    return t[-1]
```

Dice Probabilities

We wish to find the probability that rolling k 6-sided fair dice will result in a sum S. Devise an algorithm to find this probability.

```
def dice_probs(num_dice, total):
    # probs stores the probability of having a certain sum
    probs = collections.defaultdict(float)
    probs[0] = 1.0

    for i in range(num_dice):
        new_probs = collections.defaultdict(float)
        for prior_total, prob in probs.iteritems():
            # for each previous sum, we have 6 possible rolls
            for roll in range(1, 7):
                new_total = prior_total + roll
                if new_total <= total:
                    new_probs[new_total] += prob/6.0
        probs = new_probs

    return probs[total]
```

Knight Moves

Given an 8×8 chessboard and a knight that starts at position $a1$, devise an algorithm that returns how many ways the knight can end up at position xy after k moves. Knights move ± 1 squares in one direction and ± 2 squares in the other direction.

```
def knight_moves(end_position, num_moves):
    # num_ways stores how many ways there are to get to each reachable position
    num_ways = collections.defaultdict(int)
    num_ways[(0, 0)] = 1
    move_directions = [(1, 2), (1, -2), (-1, 2),
        (-1, -2), (2, 1), (2, -1), (-2, 1), (-2, -1)]

    for i in range(num_moves):
        new_num_ways = collections.defaultdict(int)
        for cur_row, cur_col in num_ways.keys():
            for move_row, move_col in move_directions:
                new_row, new_col = cur_row + move_row, cur_col + move_col
                # check to make sure new position stays within the board
                if new_row > 0 and new_row < 8 and new_col > 0 and new_col < 8:
                    new_num_ways[(new_row, new_col)] += num_ways[(cur_row, cur_col)]
        num_ways = new_num_ways

    return num_ways[end_position]
```

Egg Dropping

We have E eggs and n floors. Assume that all of the eggs are the same strength; if one egg breaks after dropping from floor i, all eggs will break after dropping from floor i. We want to find the minimum number of drops needed to find the highest floor in which an egg will not break after dropping. Devise an algorithm that returns the number of drops needed to accomplish this task.

```
def egg_drops(eggs, floors):
    # table[e][f] stores how many drops we need if we have e eggs and f floors
    table = [[0 for _ in range(floors+1)] for _e in range(eggs+1)]
    for e in range(1, eggs+1):
        table[e][1] = 1 # one floor requires just one drop

    for f in range(1, floors+1):
        table[1][f] = f # only have 1 egg, must consecutively drop upward
        table[0][f] = float('inf') # 0 eggs is impossible

    for e in range(2, eggs+1):
        for f in range(2, floors+1):
            # we should drop from the floor that minimizes the worst case number of drops
            # drop_floor signifies the floor we drop at to get to smaller problems
            # the max here goes over these two cases:
            # 1.doesn't break, same egg count, search floors above
            # 2. breaks, we lose one egg, search floors below
            drops_per_floor = [max(table[e][f-drop_floor], table[e-1][drop_floor-1])
                for drop_floor in range(1, f+1)]
            table[e][f] = 1 + min(drops_per_floor)

    return table[eggs][floors]
```

Box Stacking

We have a series of boxes, where box i has dimensions $h_i \times w_i \times d_i$. We want to create the tallest structure possible, where a box a can only be stacked on top of another box b if 2D surface of a fits strictly within the surface in contact with b. You are allowed to rotate boxes as you please. You may use each box as many times as you wish.

```
def tallest_box_stack(boxes):
    # Input: boxes has tuples of h, w, d
    # First, we make 3 copies (each surface) of each box as (w, d, h) where w <= d
    areas_and_boxes = []
    for box in boxes:
        a, b, c = box
        orientations = [sorted(b[:2]) + [b[2]] for b in [(a,b,c), (a,c,b), (c,b,a)]]
        areas_and_boxes += [(b[0]*b[1], b) for b in orientations]

    # Sort these boxes by decreasing area
    # a box with larger area cannot be stacked on top of a box with smaller area
    areas_and_boxes = sorted(areas_and_boxes)[::-1]

    # Define our dynamic programming array as the tallest stack
    # we can get by using areas_and_boxes[:i+1], where we must use box i
```

```
    max_heights = [box[-1] for area, box in areas_and_boxes]
    for i, (area, box) in enumerate(areas_and_boxes):
        for j in range(i):
            # check each previous box to see if we can stack on top of it
            below_area, below_box = areas_and_boxes[j]
            if below_box[0] > box[0] and below_box[1] > box[1]:
                max_heights[i] = max(max_heights[i], box[-1] + below_box[-1])

    return max(max_heights)
```