Problem Set 9

All parts are due on November 22, 2019 at 6PM. Please write your solutions in the LaTeX and Python templates provided. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. Solutions should be submitted on the course website, and any code should be submitted for automated checking on `alg.mit.edu`.

Please solve each of the following problems using **dynamic programming**. For each problem, be sure to define a set of subproblems, relate the subproblems recursively, argue the relation is acyclic, provide base cases, construct a solution from the subproblems, and analyze running time. Correct but inefficient dynamic programs will be awarded significant partial credit.

For each problem below, please indicate whether the requested running time is either:
(1) **polynomial**, (2) **pseudo-polynomial**, or (3) **exponential** in the size of the input.

Problem 9-1. [15 points] Career Fair Optimization

Tim the Beaver always attends the career fair, not to find a career, but to collect free swag. There are n booths at the career fair, each giving out one known type of swag. To collect a single piece of swag from booth i, having integer coolness c_i and integer weight w_i, requires standing in line at that booth for integer t_i minutes. After obtaining a piece of swag from one booth, it will take Tim exactly 1 minute to get back in line at the same booth or any other. Tim's backpack can hold at most weight b in swag; but at any time Tim may spend integer h minutes to run home, empty the backpack, and return to the fair, taking 1 additional minute to get back in a line. Given that the career fair lasts exactly k minutes, describe an $O(nbk)$-time algorithm to determine the maximum total coolness of swag Tim can collect during the career fair.

Problem 9-2. [15 points] Protein Parsing

Prof. Leric Ander's lab performs experiments on DNA. After experimenting on any **strand of DNA** (a sequence of nucleotides, either `A`, `C`, `G`, or `T`), the lab will cut it up so that any useful protein markers can be used in future experiments. Ander's lab has compiled a list P of known protein markers, where each **protein marker** corresponds to a sequence of at most k nucleotides. A **division** of a DNA strand S is an ordered sequence $D = (d_1, \ldots, d_m)$ of DNA strands, where the ordered concatenation of D results in S. The **value** of a division D is the number of DNA strands in D that appear as protein markers in P. Given a DNA strand S and set of protein markers P, describe an $O(k(|P|+k|S|))$-time algorithm to determine the maximum value of any division of S.

Problem 9-3. [15 points] **Lazy Egg Drop**

The egg drop problem presented in Lecture 17 asked for the minimum number of drops needed to determine the breaking floor of a building with n floors using at most k eggs, where the **breaking floor** is the lowest floor from which an egg could be dropped and break. If the building does not have an elevator, one might instead want to minimize the **total drop height**: the sum of heights from which eggs are dropped. Suppose each of the n floors of the building has a known positive integer height h_i, where floor heights strictly increase with i. Given these heights, describe an $O(n^3k)$-time algorithm to return the minimum total drop height required to determine the breaking floor of the building using at most k eggs.

Problem 9-4. [15 points] **Alien Chopsticks**

Aliens have heard of the human game called chopsticks[1], and want to start playing as well. Each alien has two hands, but each alien may have a different number of fingers on each of their hands. The two-player game of **alien chopsticks** works as follows.

- Each alien player starts the game by lifting exactly one finger on each hand.
- Then the two aliens take turns making **moves**.
- To make a move, an alien:
  - chooses one of their own hands, currently lifting i fingers, and
  - touches one of their opponent's hands currently lifting j fingers.
  - The opponent's hand must lift i more fingers, in addition to the j fingers already lifted.
  - If the opponent's hand has fewer than $i + j$ fingers, that hand is **disabled**.
  - A disabled hand cannot participate in any moves.
  - A player loses when both of their hands are disabled.

Describe an $O(abcd)$-time algorithm to determine whether an alien with a fingers on its left hand and b fingers on its right hand who **starts first**, can always force a win against an alien with c fingers on its left hand and d fingers on its right.

[1] `https://en.wikipedia.org/wiki/Chopsticks_(hand_game)`

Problem 9-5. [40 points] **Building a Wall**

The pigs in Porkland from Problem Set 2, have decided to build a stone wall along their southern border for protection against the menacing wolf. The wall will be one meter thick, n meters long, and at most k meters tall. The wall will be built from a large supply of identical **long stones**: each a $1 \times 1 \times 2$ meter rectangular prism. Long stones may be placed either vertically or horizontally in the wall. With much difficulty, a single long stone can be broken into two 1-meter **cube stones**, but the pigs prefer not using cube stones when possible.

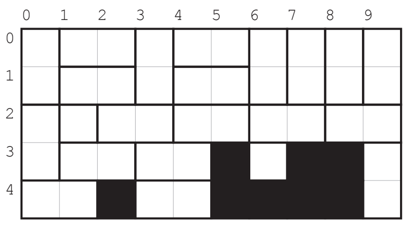
The ground along the southern border of Porkland is uneven, but the pigs have leveled each square meter along the border to an integer meter elevation. Let a **border plan** be an $n \times k$ array `B` correspond to what the border looks like before a wall has been built. `B[j][i]` corresponds to the cubic meter whose top is at elevation $k - j$, located at meter i along the border. `B[j][i]` is `'.'` if that cubic meter is **empty** and must be covered by a stone, and `'#'` if that cubic meter is **dirt**, so should not be covered. `B` has the property that if `B[j][i]` is covered by dirt, so is every cubic meter `B[t][i]` beneath it (for $t \in \{j, \dots, k-1\}$), where the top-most cubic meter `B[0][i]` in each column is initially empty. Below is an example `B` for $n = 10$ and $k = 5$.

A **placement** of stones into border plan B is a set of placement triples:

- `(i,j,'1')` places a cube stone to cover `B[j][i]`;
- `(i,j,'D')` places a long stone oriented down to cover `B[j][i]` and `B[j + 1][i]`; and
- `(i,j,'R')` places a long stone oriented right to cover `B[j][i]` and `B[j][i + 1]`.

A placement is **complete** if every empty cubic meter in `B` is covered by some stone; and is **non-overlapping** if no cubic meter is covered by more than one stone and no stone overlaps dirt. Below is a complete non-overlapping placement for `B` that uses 2 cube stones, and a pictorial depiction.

```
B = [                         P = [
      '..........',                 (0,0,'D'), (0,2,'D'), (0,4,'R'), (1,0,'R'), (1,1,'R'),
      '..........',                 (1,2,'1'), (1,3,'R'), (2,2,'R'), (3,0,'D'), (3,3,'R'),
      '..........',                 (3,4,'R'), (4,0,'R'), (4,1,'R'), (4,2,'R'), (6,0,'D'),
      '.....#.##.',                 (6,2,'R'), (6,3,'1'), (7,0,'D'), (8,0,'D'), (8,2,'R'),
      '..#..####.',                 (9,0,'D'), (9,3,'D'),
]                             ]
```



(a) [15 points] Given $n \times k$ border plan B, describe an $O(2^{2k}kn)$-time algorithm to return a complete non-overlapping placement for B using the fewest cube stones possible.

(b) [25 points] Write a Python function `build_wall(B)` that implements your algorithm from (a) **for border plans with $k = 5$**. You can download a code template containing some test cases from the website. Submit your code online at `alg.mit.edu`.