Problem Session 9

Problem 9-1. Coin Crafting

Ceal Naffrey is a thief in desperate need of money. He recently acquired n identical gold coins. Each coin has distinctive markings that would easily identify them as stolen if sold. However, using his amateur craftsman skills, Cael can melt down gold coins to craft other golden objects. Ceal has a buyer willing to purchase golden objects at different rates, but will only purchase one of any object. Ceal has compiled a list of the n golden objects, listing both the positive integer **purchase price** the buyer would be willing to pay for each object and each object's positive integer **melting number**: the number of gold coins that would need to be melted to craft that object. Given this list, describe an efficient algorithm to determine the maximum revenue that Ceal could make, by melting down his coins to craft into golden objects to sell to his buyer.

Solution:

1. **Subproblems**
   - Label each craft-able object with a unique integer from 1 to n
   - Let p_i be the purchase price of object j, with k_i its melting number
   - $x(i, j)$: the maximum revenue possible from i coins, being able to craft any of the objects from the objects from 1 to j
2. **Relate**
   - Guess whether or not to craft object j
   - If $i < k_i$, object j cannot be crafted
   - If object j is not crafted, may recurse on remaining items
   - If object j is crafted, receive p_i in revenue and lose k_i coins
   - $x(i,j) = \begin{cases} x(i, j-1) & \text{if } i < k_i \\ \max(p_i + x(i-k_i, j-1), x(i, j-1)) & \text{otherwise} \end{cases}$
3. **Topo. Order**
   - Subproblem $x(i, j)$ only depends on subproblems with strictly smaller j, so acyclic
4. **Base**
   - If there are not more coins, or no more items, cannot gain any revenue
   - $x(0, j) = 0$ for $i \in \{0, \ldots, n\}$
   - $x(i, 0) = 0$ for $j \in \{0, \ldots, n\}$
5. **Original**
   - $x(n, n)$ is the maximum revenue possible from n coins, being able to craft any of the $n$objects, as requested
6. **Time**
   - # subproblems: $(n+1)^2 = O(n^2)$, $x(i, j)$ for $i, j \in \{0, 1, \ldots, n\}$
   - work per subproblem: $O(1)$
   - $O(n^2)$ running time

Problem 9-2. 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.

Solution:

1. **Subproblems**
   - $x(i, j)$: the maximum total coolness of swag collectible in the next i minutes with j weight remaining in Tim's backpack (where Tim is at the career fair and may immediately stand in line)
2. **Relate**
   - Tim can either:
     - Collect no more swag
     - Stand in a line and collect a piece of swag
     - Go home and empty the backpack
   - $x(i,j) = \max \begin{cases} 0 & \text{always} \\ c_{k'} + x(i - t_{k'} - 1, j - w_{k'}) & \text{for } k' \in \{1, \dots, n\} \text{ where } w_{k'} \le j, t_{k'} < i \\ x(i - h - 1, b) & \text{when } i > h \end{cases}$
3. **Topo. Order**
   - Subproblem $x(i, j)$ only depends on subproblems with strictly smaller i, so acyclic
4. **Base**
   - Tim needs time to collect swag, so nothing possible if time is zero
   - $x(0, j) = 0$ for $j \in \{0, \dots, b\}$
5. **Original**
   - $x(k, b)$ is the maximum total coolness of swag collectible in k minutes, starting with an empty backpack, as desired.
6. **Time**
   - # subproblems: $\le (k + 1)(b + 1) = O(kb)$, $x(i, j)$ for $i \in \{0, \dots, k\}, j \in \{0, \dots, b\}$
   - work per subproblem: $O(n)$
   - $O(nbk)$ running time, which is **psuedo-polynomial** in b and k

Problem 9-3. 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 $D = (d_1, \dots, d_m)$ 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.

Solution:

1. **Subproblems**
   - First construct a hash table containing the protein markers in P as keys
   - This hash table takes (expected) $O(k|P|)$ time to construct (it could take $O(k)$ time to compute the hash of each marker)
   - $x(i)$: the maximum value of any division of the DNA strand suffix $S[i:]$
2. **Relate**
   - Either the first nucleotide starts a protein marker or it does not
     - If it does not, recurse on remainder
     - If it does, guess its length (from 1 to k, or until the end of S)
   - Let $m(i, j)$ be 1 if substring $S[i : j]$ is a protein marker and 0 otherwise
   - We can evaluate $m(i, j)$ in expected $O(k)$ time by hash table look up
   - $x(i) = \max\{m(i, j) + x(i + j) \mid j \in \{1, \ldots, \min(k, |S| - i)\}\}$
3. **Topo. Order**
   - Subproblem $x(i)$ only depends on subproblems with strictly larger i, so acyclic
4. **Base**
   - If no nucleotides in suffix, no division can have value
   - $x(|S|) = 0$
5. **Original**
   - $x(0)$ is the maximum value of any division of S, as desired
6. **Time**
   - # subproblems: $\leq |S| + 1 = O(|S|)$, $x(i)$ for $i \in \{0, \ldots, |S|\}$
   - work per subproblem: expected $O(k)$ to look up each of the k $m(i, j)$
   - $O(k(|P| + k|S|))$ running time, which is **polynomial** in the size of the input
   - Note that it is possible to achieve $O(k(|P| + |S|))$ if each of the $\Theta(k|S|)$ lookups can be checked in $O(1)$ time, i.e., via a rolling hash or suffix tree (not covered in this course).

Problem 9-4. Lazy Egg Drop

The classic egg drop problem asks 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. This problem has a closed form solution, but can also be solved with dynamic programming (Exercise!). However, 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.

Solution:

1. **Subproblems**
   - $x(i, j, e)$ minimum total drop height needed to determine the breaking floor with e eggs, with floors i to $j \geq i$ remaining to check
2. **Relate**
   - Next egg must be dropped from some remaining floor f (guess!)
   - If egg breaks, have one fewer egg and need to recurse on floors below
   - Otherwise, have same number of eggs and need to recurse on floors above
   - $x(i, j, e) = \min\{h_f + \max\{x(i, f-1, e-1), x(f+1, j, e)\} \mid f \in \{i, \ldots, j\}\}$
3. **Topo. Order**
   - Subproblem $x(i, j, e)$ only depends on subproblems with strictly smaller $j - i$, so acyclic
4. **Base**
   - Impossible if no more eggs but floors left to check
   - $x(i, j, 0) = \infty$ for $i, j \in \{0, \ldots, n\}$ with $i \leq j$
   - No more drops needed if no more floors left to check
   - $x(i, i-1, e) = 0$ for $i \in \{0, \ldots, n\}$ and $e \in \{0, \ldots, k\}$
5. **Original**
   - $x(1, n, k)$ is the minimum total drop height needed to determine breaking floor with k eggs needing to check all floors, as desired
6. **Time**
   - # subproblems: $\leq \binom{n}{2}(k+1) = O(n^2 k)$, $x(i, j, e)$ for $i, j \in \{1, \ldots, n\}$ with $i < j$ and $e \in \{0, \ldots, k\}$
   - work per subproblem: $O(\max(1, j - i)) = O(n)$ time
   - $O(n^3 k)$ running time, which is **psuedo-polynomial** in k. However, since you will never need to drop more than n eggs to check all n floors, it suffices to only compute subproblems for $k \leq n$, so we could answer the problem in $O(n^3)$-time by limiting the subproblems checked, which is polynomial in n.

Problem 9-5. Building a Wall

The pigs in Porkland from Problem Session 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 $t \in \{j, \ldots, k-1\}$), where

the top-most cubic meter `B[0][i]` in each column is initially empty. Below is an example `B` for $n = 10$ and $k = 5$.

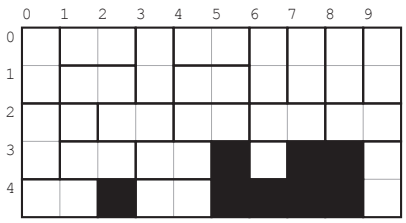
A **placement** of stones into border plan B is a set of placement triples:

- `(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) 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.

Solution:

1. **Subproblems**
   - The approach will be to repeatedly cover the highest uncovered cube in the left-most column with some stone
   - Placing a single stone will effect at most the first two columns, so our subproblems will remember the current covered state of the two left-most columns
   - Represent a **partially-filled column** c of the border as a length-k array of Boolean values, where Boolean $c[j]$ is True if row j in the column still needs to be covered, and False otherwise
   - Let $C_0(i)$ correspond to the partially-filled column i in the input border plan `B`, where $C_0(i)[j]$ is True if `B[j][i] = '.'` and False if `B[j][i] = '#'`
   - Let $C_0(n) = C_0(n+1) = c_F$ correspond to a column of all False values
   - $x(i, c_1, c_2)$: the minimum number of cube stones needed to cover partially-covered columns c_1 and c_2 concatenated with original columns $i + 2$ to $n - 1$
2. **Relate**
   - If all rows of the left-most column are covered, we can move on to cover uncovered cubes in the next column

- Otherwise, place a stone to cover the highest uncovered cube in the left-most column with either:
  - a cube stone;
  - a long stone vertically down (if the cube below is also uncovered); or
  - a long stone horizontally right (if the cube to the right is also uncovered).
- Let $f(c, k)$ correspond to the partially filled column resulting from changing element $c[k]$ in column c to False
- Let $t(c)$ correspond to the index of the first (top-most) True in column c
- $x(i, c_1, c_2) = x(i + 1, c_2, C_0(i + 2))$ if c_1 is all False
- Otherwise, $t(c_1)$ is defined:
- $x(i, c_1, c_2) = \min \left\{ \begin{array}{ll} 1 + x(i, f(c_1, t(c_1)), c_2) & \text{always} \\ x(i, f(f(c_1, t(c_1)), t(c_1) + 1), c_2) & \text{if } c_1[t(c_1) + 1] \text{ is True} \\ x(i, f(c_1, t(c_1)), f(c_2, t(c_1))) & \text{if } c_2[t(c_1)] \text{ is True} \end{array} \right\}$

3. **Topo. Order**
   - Subproblem $x(i, c_1, c_2)$ either strictly increases i or strictly decreases the number of True values appearing in column c_1 for the same i, so acyclic
4. **Base**
   - No cube stones are required when nothing remains to be covered
   - $x(n, c_F, c_F) = 0$
5. **Original**
   - $x(0, C_0(0), C_0(1))$ corresponds to the minimum number of cube stones to fill the entire boarder plan, as desired
   - Store parent pointers to reconstruct an optimizing placement
6. **Time**
   - There are 2^k possible columns for each of c_1 and c_2
   - # subproblems: $\leq n2^k2^k = O(n2^{2k})$ subproblems
   - work per subproblem: $O(k)$ time to lookup in memo and/or to compute a new column
   - $O(2^{2k}kn)$ running time, which is **exponential** in k (but polynomial in n)

(b) 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`.

Solution:

```
def get_col(B, i):
    if i < len(B[0]):
        return tuple(B[j][i] == '.' for j in range(5))
    return tuple(False for j in range(5))

def build_wall(B):
    memo = {}
    def x(i, c1, c2):                       # top-down dynamic program
        key = (i, c1, c2)
        if key not in memo:                 # compute if not in memo
            j = 0
            while j < 5 and (not c1[j]):
                j += 1
            if j == 5:                      # column c1 is full
                if i == len(B[0]) - 1:      # base case, last line
                    memo[key] = (0, None, None)
                else:                       # shift to next pair of cols
                    memo[key] = x(i + 1, c2, get_col(B, i + 2))
            else:                           # need to cover (i, j)
                new_c1 = [i for i in c1]
                new_c1[j] = False
                test, _, _ = x(i, tuple(new_c1), c2)  # place 1x1
                best = 1 + test
                move = (i, j, '1')
                parent = (i, tuple(new_c1), c2)
                if j < 4 and c1[j + 1]:             # place 1x2 down
                    new_c1[j + 1] = False
                    test, _, _ = x(i, tuple(new_c1), c2)
                    if test < best:
                        best = test
                        move = (i, j, 'D')
                        parent = (i, tuple(new_c1), c2)
                    new_c1[j + 1] = True
                if c2[j]:                           # place 1x2 right
                    new_c2 = [i for i in c2]
                    new_c2[j] = False
                    test, _, _ = x(i, tuple(new_c1), tuple(new_c2))
                    if test < best:
                        best = test
                        move = (i, j, 'R')
                        parent = (i, tuple(new_c1), tuple(new_c2))
                memo[key] = (best, move, parent)
        return memo[key]
    best, move, parent = x(0, get_col(B, 0), get_col(B, 1))
    P = []              # compute placement from parent pointers
    while parent:
        P.append(move)
        best, move, parent = memo[parent]
    return P
```