Dynamic Programming Algorithms

1. (a) Let $T(i)$ denote the number of ways to encode the string up to and including $S[i]$.

$$T(i) = \begin{cases} 1 & i = -1 \\ 1 & i = 0 \text{ and } S[i] \in \{1, \ldots, 9\} \\ T(i-1) & i \geq 1 \text{ and } S[i] \in \{1, \ldots, 9\} \text{ and } S[i-1:i+1) \notin \{10, \ldots, 26\} \\ T(i-1) + T(i-2) & i \geq 1 \text{ and } S[i-1:i+1) \in \{10, \ldots, 26\} - \{10, 20\} \\ T(i-2) & i \geq 1 \text{ and } S[i-1:i+1) \in \{10, 20\} \\ 0 & \text{otherwise} \end{cases}$$

   (b) Let $P(k, s)$ denote the probability of k dice rolls summing to s.

$$P(k, s) = \begin{cases} 1 & k = 0 \text{ and } s = 0 \\ 0 & k = 0 \text{ and } s \neq 0 \\ \frac{1}{6} \sum_{i=1}^{6} P(k-1, s-i) & k > 0 \end{cases}$$

   (c) Let $N(k, \hat{x}, \hat{y})$ denote the number of ways for a knight to end up on space $\hat{x}\hat{y}$ after k moves and $R(\hat{x}, \hat{y})$ be the set of reachable spaces from space $\hat{x}\hat{y}$.

$$N(k, \hat{x}, \hat{y}) = \begin{cases} 0 & k = 0 \text{ and } \hat{x}\hat{y} \neq A1 \\ 1 & k = 0 \text{ and } \hat{x}\hat{y} = A1 \\ \sum_{(x', y') \in R(\hat{x}, \hat{y})} N(k-1, x', y') & \end{cases}$$

2.
```
def box_stacking(boxes):
    """
    Inputs:
    - boxes: Tuples of (h, w, d)
    """
    # Makes 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]

    # Sorts these boxes by decreasing area
    areas_and_boxes = sorted(area_and_boxes)[::-1]

    # Defines our DP array as the tallest stack
    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):
```

```
    # checks each previous box to see if we can stack on top of it
    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)
```

3. (a) We are interested in maximizing the value on day 2 of the stocks that we purchase on day 1. Given a budget of P, we can compute the maximum profit, $M[l]$ attainable over the k stocks while spending l dollars, for $l \in [1, P]$. Note that for a given l,

$$M[l] = \max_{j \in [0,k-1], p_{0,j} \leq l} \{(p_{1,j} - p_{0,j}) + M[l - p_{0,j}]\}$$

In other words, if we have l dollars to spend, we can maximize our profits by considering the maximum profit attainable by purchasing a unit of the jth stock as well as the maximum profit attainable with a budget of l minus the price of the jth stock.

We maintain a $1 \times (P+1)$ array M of profits for each price $l \in [0, P]$. Our algorithm proceeds as follows:

- Initializes $M[0] = 0$.
- For each l, assuming $M[l']$ is computed correctly for all $l' \leq l$, computes $M[l]$ according to the equation above by iterating over the k stocks.
- Finally, passes over M and look for the entry of maximum value: this is the maximum attainable profit by investing the money available.

The algorithm iterates over $O(P)$ budget allowances and, for each budget allowance, iterates over $O(k)$ stocks. Then, it iterates over list M one more time. Thus, the algorithm runs in $O(kP)$-time.

(b) To find the optimal investment strategy over n days, consider applying our strategy for two days at each interval. If we ever discover that our earnings exceed Q dollars, we can return success immediately. Otherwise, we have to continue running our algorithm to compute the profits attainable from day i to day $i+1$ (and add these profits to our starting budget) to obtain our budget for the $i + 1^{\text{st}}$ day. If after n days, we never obtained Q dollars, we can report the maximum earnings we can make, by summing P and the total maximum profits.

To see why this procedure produces the optimal profits, suppose that in the optimal investment strategy, we purchase a stock on day i and sell it on day j. Because we are allowed to buy and sell as many stocks as we want per day without cost, this is equivalent to purchasing the stock on day i, selling it on day i+1, then purchasing on day i+1 and selling on day i+2, etc. until we buy on day j-1 and sell on day j. Thus, at each step, we only need to maximize the total for the next day.

Each of our budgets before success will be some $P < Q$. Thus, we can bound the running time of each of the n-1 iterations by $O(kQ)$. Thus, the algorithm runs in $O(nkQ)$-time.

Greedy Algorithms

---

1. Define $T = V - U$. Create an MST out of the nodes in T (using Prims's algorithm for example). Then, add vertices in U to this tree by taking the lightest edge from a node $u \in U$ to another node $t \in T$.

2. Consider a graph G constructed as follows. We have a vertex $v_i \in V$ for $i = 1, \ldots, n$, one for each of the n cities, and there is an edge between v_i and v_j with cost $c_{i,j}$. Consider also the graph G_{sky}, which is the same as G and where we additionally add a vertex `sky` $\in V$ representing the sky. We add an edge between v_i and `sky` with cost a_i. Now our algorithm is:

```
planTransit(costs c_{i,j}, a_i):
  generate the graph G as above
  T = PRIMS(G) # or any MST algorithm
  T_sky = PRIMS( G_sky )
  if cost(T) < cost(T_sky):
    T_actual = T
  else:
    T_actual = T_sky
  airports = []
  roads = []
  for i = 1, ..., n:
    if {v_i, air} is an edge in T_actual:
      airports.add(i)
  for j = 1, ..., n:
    if {v_i, v_j} is an edge in T_actual:
      roads.add( {i,j} )
  return airports, roads
```

3. (a) Take the largest and smallest values. This guarantees the largest unfairness value because we have the largest difference we could possibly generate with the pieces available.

   (b) First, sort the pieces by size. Then, slide a window of size k over the sorted array, and check the difference between the largest and smallest values in the window. Finally, return the window with the smallest difference.