CS 344 Homework Two

Matthew McCaughan

Due February 27th, 2025

Problem 1: Academic Integrity

I affirm that I have not given or received any unauthorized help on this assignment, and that this work is my own.
- Matthew McCaughan

Problem 2: Cycle Clustering

Similar to our 1-Dimensional k-means clustering from lecture, we'll be leveraging dynamic programming to formulate an algorithm for our cycle clustering problem. The arc length will be the analogue for distance between points on the circle. I'll define a function that will minimize our target quantity from target integer k and points from x_1 to x_v

$$F(k, v) = \text{Cost of best k-clustering solution on prefix } x_1 \text{ to } x_v$$

To minimize:

$$\sum_{i=1}^{n} \min_{j \in [k]} d(x_i, c_j)$$

I'll assume the base case of our function to contain zero points from the problem, or just one cluster. Other cases will be more complicated. We'll define these as follows:

$$F(k,v) = \begin{cases} 0, & \text{if v} = 0 \\ \sum_{i=1}^{n} \min_{\text{direction}}((d(x_1, x_k \text{ clockwise}), d(x_1, x_k \text{ counterclockwise}))), & \text{if k} = 1 \\ \sum_{k=1}^{v} (\min_{k-cluster} \sum_{i=1}^{n} (\min_{\text{direction}}((d(x_1, x_k \text{ clockwise}), d(x_1, x_k \text{ counterclockwise}))))), & \text{else} \end{cases}$$

When v is zero, we have no points selected from the available points in the problem, so regardless of the amount of clusters, there will be no distance to measure.
When $k = 1$, we have only one cluster chosen from any available amount of clusters k, regardless of the amount of points, for each point, the smaller distance to that cluster, either by going clockwise or counterclockwise, will be the shortest possible distance to the cluster for that point, and we do a summation through all points.
Our final case, in which most cases will occur, is similar to our single-cluster case. The difference is that for each point, for each cluster, we are get the minimum sum of the points' distances to the cluster, and within each we are choosing the direction (clockwise or counterclockwise) that minimizes that distance to each cluster we check.
Overall, we are checking the minimum distance for each direction from a point to a cluster, for every cluster, finding that minimum sum of distances, and finding the minimum of these sums. Checking every state and finding the minimums among each component will guarantee a solution. If we assume that there is a less than ideal solution to each component of the minimization, the algorithm would've found a better solution for the problem.

There exists $n * k$ states that these points can be attributed to the clusters, and we are iterating through each point to determine its minimum distance, which is at most 0 to n points. Giving us an approximate run time in the order of $O(n^2 * k)$, which matches our 1-dimensional clustering case. I think this makes an intuitive sense because the circle can be split at a point and be flattened out to be modeled similarly to our 1-dimensional. Though likely less ideal of a solution, the circle can be split and flattened at an arbitrary point x_i and have it flattened into a line to give a different view of the problem.

Problem 3: Primeland ATM

With Primeland's bizarre choice of denominations, we can formulate a solution to count the number of ways to split our target n into the denominations. With a minimum target $n \geq 2$, $n = 2$ will form our base case for the function we can define, $F(n)$. For values of $n \geq 2$, the highest denomination that is less than or equal to n will either be included in the solution or will not be included in the solution, this forms two cases for each call.

$$F(n) = \begin{cases} 0, & \text{if } 0 \leq n \leq 2 \\ 1, & \text{if n = 2} \\ F(\text{n - next largest denom.}) + F(\text{n - current denom.}), & \text{if n > 2} \end{cases}$$

For example if n = 3, our a solution will either use the denomination of 3, or it will not use this denomination, and use only the denominations lower than 3 (just 2), forming the cases (3) and (3-2). Only one of these cases will perfectly capture n = 3, (using the 3 coin).
Similarly, for n = 4, we will get the clases (4-2) and (4-3), only one of which will return a valid way to split the target.
We can memoize the results of our function F to significantly reduce the number of sub-computations we need to perform. Beyond our base case of F(2) = 1, every $F(n > 2)$ will utilize the current largest denomination used or will use the next largest denomination. This guarantees that at every intermediate n calculated, any way to split the target n will be exhaustively searched for.
For this problem, the algorithm will check the valid splits using each denomination, so we iterate through the list of denominations d. And for each f(n), assuming that all $n-1$ values previously are valid, we will be iterating up to n. With these two iterations taking place, this will give us an approximate run time in the order of $O(d * n)$.

Problem 4: Stop asking me to fix your hard drive

Assuming that $\sum_{i=1}$ Score(b_i) is maximized when strings $(b_1, b_2, b_3...)$ are exactly equivalent to the original data (although we do not know the original data), b_i will be maximized when the last number is appended to form the original bit string, and will be less with the next bit appended to it. We will find the maximum of these two cases for each bit we add to our current string b, and once a local maximum is reached, we start a new string starting with the next bit.
I'll prove this along the length of the original bit string from bits at position 0 to ℓ. I'm going to assume that $SCORE(b)$ returns a decimal value from 0 to one to represent a percentage similarity.

$$F(\ell) = \begin{cases} 1, & \text{if } \ell = 1 \\ \sum_{b=1}^{i} \max(\text{SCORE(b formed up to } \ell),(\text{SCORE(b formed up to } \ell+1), & \text{if } \ell > 1 \end{cases}$$

At bit position ℓ, the max SCORE(b_i) is achieved either with the string b_i including bit at position ℓ, or the string including the bit at position $\ell + 1$. Once a local maximum is discovered by iterating through ℓ, we end the string b_i and begin string $b_i + 1$ starting with the bit at position $\ell + 1$. Based on this rigorous check, we know that each bit string b_i is maximized for score, therefore the set of strings b_1 to b_i will be maximized for the score.
If we save the computation for $F(\ell)$ as we go, we only calculate F for each length of the original bit list once, resulting in an approximate running time of $O(n)$. We must go through at least the length of the original bit list to guarantee a maximized score for each bit list b_i

Problem 5: Space Lasers

The primary choice to make for this problem is if the time spent d with the loss of signal s on the original satellite is worth the time t spent with the new satellite's signal s'. We are dependent on the number of satellites and the $SIGNAL(s_i, t)$ for satellite i. We can model the different cases for our function that will maximize $\sum_{t=1}^{T} SIGNAL(s, t)$, with a new function F(s,t). s' will denote a new satellite that is switched to.

$$F(s,t) = \begin{cases} max_s \text{ Signal(s,1)}, & \text{if t} = 1 \\ \max((\sum_{t}^{d} SIGNAL(s,t)), SIGNAL(s', t+d)), & \text{if } t > 1 \end{cases}$$

In the case of only one satellite s_1, we still start with the satellite and not change, so no need to worry about switching, and our max will be $\sum_{t=1}^{T} SIGNAL(s_1, t)$. For numbers of satellites greater than 1, we have to check if switching to another satellite to get signal s' (s' can be among all of the other satellites) after spending time d with a signal quality 0 will result in a greater overall signal value than if we remained with our original satellite from time t up to time d. If we get a higher signal score from having stayed with our original satellite, then we remain pointed at it.
Initially, we check the signal score of all satellites to start at time $t = 1$, for every subsequent time t we check our signal score against switching to the $s - 1$ other satellites, so we check our $SIGNAL(s, t)$ for every satellite for every time t. This in total gives us a running time of approximately $O(s^2 * t)$, where s is the number of different satellites and t is the range of time we are measuring.

Problem 6: Longest Increasing Double Subsequence

To find the longest Increasing subsequence that is a subsequence of both X and Y, we can perform the longest increasing subsequence algorithm on one sequence (X or Y), while subsequently checking if the other sequence has the same subsequence and is strictly increasing. We can simply check if the same values both appear with increasing indices. My algorithm is defined in pseudocode as follows:

C = 0 (value to track length of subsequence)
For value in X:

if There exists a value V in both X and Y **then**
C = 1
while There exists a value $V' > V$ in both X and Y **do**
if $(X[V'] > X[V])$ && $(Y[V'] > Y[V])$ **then** (W[Z] denotes index of Z in a list W)
C = C+1 (increment length counter)

We want to return $max(C)$ among all values in X. $max(C)$ will return the longest length of values in common between X and Y, that within each sequence X or Y have both values and indices that are strictly increasing from the current position in the substring to the next. I'll make an example to model my solution:

$$X = (1, 2, 1, 0, 1) \text{ and } Y = (9, 6, 3, 1, 2)$$

Starting with value one in X, we check to see value one exists in both X and Y (which is does), and then we check to see while there is a value, if there is a value greater than one that occurs in both X and Y (which is 2), and also if this value occurs at an index greater than 0 in X, and 3 in Y. We will increment our length tracker in this case. The algorithm will break at the while loop after we've checked that no other values between the lists have a greater value than the current highest value and respectively have a greater index in each sequence. The longest increasing double subsequence is 2 in this example.
We will increment through the length of sequence X, and in a legitimate implementation, would need to increment through the sequence of Y to check for matching values and valid indices. These nested checks of each sequence would result in a run time of approximately $O(x * y)$, where x is the length of sequence X and y is the length of sequence y.