# Data Structures and Algorithms Homework 13

Due Wednesday Dec 4; Joseph Sepich (jps6444)

## 1 Problem 1 Cross-Country Trip

Philbert is going on a cross-country trip. He is starting at mile marker zero and can stop at n hotels located at a location/mile maker denoted $h_i$. We want to minimize total penalty, which occurs when Philbert does not meet his max miles per day. The cost function for a single day is denoted (150 - m)^2 where m is the number of miles travelled. This problem can be cast as a dynamic pogramming problem. Below is the algorithm to optimize his problem:

```
Min_Penalty(array H) {
  totalPenalty = new array P[|H|] // create cost tracking array
  foreach p in P {
    P[p] = MAX_INTEGER // initialze cost to infinity
  }

  for (i in 1:|H|) {
    if (i == 1) {
      if (H[i] > 150) // problem not solvable {
        return NULL
      }
      P[i] = (150 - H[i])^2 // initial penalty
    }
    P[i] = findCost(H[i], H, P)
  }
  return P[len(P) - 1] // last element is destination and total cost
}

findCost(int h,array H, array P) { // looks at previous subproblems and finds new min
  costs = new array C[]
  foreach hotel < h in H {
    if (h - hotel > 150) {
      continue // not in range
    } else {
      diff = h - hotel // miles travelled
      totalCost = (150 - diff)^2 + P[hotel] // cumulative penatly
      costs.Add(totalCost)
    }
  }
  return min(costs) // min
}
```

Collaborators: None

## 2 Problem 2 LPS

We want to find the longest palindromic subsequence. The algorithm should run in $O(n^2)$ time. This implies a dynamic programming approach to the problem. Using this approach we can see that in order to find a longest subsequence that includes a[n], we would need to compute the longest subsequence of its predecessors. It could not be a[n] alone, because then any of the elements would be a viable answer if nothing longer than one element exists. In order to determine if the subsequence is a palindrome, we need to know if it is the same sequence when read in reverse. To do this we can have the original and reversed strings, and the largest common sequence would be the longest palindromic subsequence of the original sequence.

```
LPS(array a) { // a[1..n] is our original sequence
  R = a.reverse()
  sequence = new list[]
  while (!len(R) == 0 &&  !len(a) == 0) {
    lastR = R.pop()
    lasta = a.pop()
    if (lastR != lasta && (len(R) == 0 || len(a) == 0)) { // no matching letters left
      return sequence
    }
    if (lastR == lasta) {
      sequence.add(lastR)
      continue
    } else {
      a.add(lasta)
      R.add(lastR)
      max(len(LPS(a)), len(LPS(R))) // figure out which subsequence is better through recursion
    }
  }
  return sequence
}
```

## 3 Problem 3 Change Making

For this change making problem we can use a matrix approach to finding values that work for possible change. This can work, since the solution runs in O(nv) time, we are using the i rows as our coins, and j columns as our values. The base case would be the 0 column, which works with 0 coins.

```
Change(array X, int v) {
  V = 0:v
  bool[][] possible = new bool[|X|][|V|] // false by default
  possible[0:len(X)][0] = true // base case

  for (int i=0; i < len(X); i++) { // go through each coin value
    for (int j=1; j < len(V); j++) { // go through each value
      // check previous coins are possible
      if (possible[i-1][j] == true) {
        possible[i][j] = true
      } else if (possible[i][j-X[j]])  {
        // check if possible by adding current coin
        // this adds a duplicate
        possible[i][j] = true
      }
    }
  }
  return possible[len(X)][len(V)]
}
```

## 4 Problem 4 Change Making More

This problem is the same as the previous, BUT each coin can only be used once. We merely need to make a modification to the previous algorithm to fix this. The best way to ensure this is by using a two dimensional table. Instead of merely checking to see if the previous value was possible, we now have to also check what coins we are using.

```
Change(array X, int v) {
  V = 0:v
  bool[][] possible = new bool[|X|][|V|] // false by default
  possible[0:len(X)][0] = true // base case

  for (int i=0; i < len(X); i++) { // go through each coin value
    for (int j=1; j < len(V); j++) { // go through each value
      // check previous coins are possible
      if (possible[i-1][j] == true) {
        possible[i][j] = true
      } else if (possible[i-1][j-X[i]])  {
        // check if possible by adding current coin
        // LOOKS TO SEE IF YOU CAN ADD TO PREVIOUS COIN
        possible[i][j] = true
      }
    }
  }
  return possible[len(X)][len(V)]
}
```

## 5 Problem 5 Change Making Again

In this last coin change problem we wish to know if change is possible using at most k coins. This is a minimization problem. If we find the minimum possible coins to make change, and that value is less than or equal to k then it is possible. So we can again use duplicate denominations, but we must track the number of coins.

```
Change(array X, int v,int k) {
  V = 0:v
  int[][] possible = new int[|X|][|V|]
  possible[0:len(X)][0] = 0 // base case
  possible[0:len(X)][1:len(V)] = MAX_INTEGER

  for (int i=0; i < len(X); i++) { // go through each coin value
    for (int j=1; j < len(V); j++) { // go through each value
      // check previous coins are possible
      prevAmount = possible[i-1][j]
      // check if possible by adding current coin
      addCoin = possible[i][j-X[j]] + 1
      addCoinPrev = possible[i-1][j-X[j]] + 1
      possible[i][j] = min(prevAmount, addCoin, addCoinPrev)
    }
  }
  return possible[len(X)][len(V)] < k
}
```