Homework 4

1. A subsequence is a *palindromic* if it is the same whether read left to right or right to left. For instance, the sequence

$$A, C, G, T, G, T, C, A, A, A, A, T, C, G$$

has many palindromic subsequences, including A, C, G, C, A and A, A, A, A (on the other hand, the subsequence A, C, T is *not* palindromic). Use dynamic programming to devise an algorithm that takes a sequence $x[1 \ldots n]$ and returns the (length of the) longest palindromic subsequence. Its running time should be $O(n^2)$.

Solution:

1. Let $Palin(i, j)$ be the length of the longest palindromic subsequence between (and including) characters i and j.

2. This is a bit like longest common subsequence. Let x be our string. Then if $x[i] = x[j]$, then these endpoints can match with each other, and the length of the longest palindrome is $2 + Palin(i + 1, j - 1)$.

   If $x[i] \neq x[j]$, then these endpoints are not both in the longest palindrome, and we may eliminate one of them without changing the longest palindrome. We don't know which, so we use $\max(Palin(i+1, j), Palin(i, j-1)$. The tricky part is that we should compute these values in order of increasing *length* of the sequence - we can't do a simple loop over all i or j. For our base case, $Palin(i, i-1) = 0$ (a zero-length string), and $Palin(i, i) = 1$, since an individual character is a palindrome.

3. (without memoization)

```
function LongestPalin(Array x[1...n])
    Array Palin[1...n][1...n]
    for i = 1 to n do
        Palin[i][i] ← 1
        Palin[i][i − 1] ← 0
                                              ▷ Initialization of values
    for l = 1 to n − 1 do                     ▷ Increasing lengths
        for i = 1 to n − l do
            j ← i + l
            if x[i] = x[j] then
                Palin[i][j] ← 2 + Palin[i + 1][j − 1]
            else
                Palin[i][j] ← max(Palin[i + 1][j], Palin[i][j − 1])
    return MaxL
```

4. (with memoization)

```
function LONGEST_PALIN(Array inputString[1...n])
    Array palinS[1...n][1...n]
    return PALINR(1, n, inputString, palinS)
function PALINR(int start, int end, inputString[1...n], palinS[1...n][1...n])
    if start == end then                      ▷ Base Case 1.
        return 1
```

if start $>$ end **then** ▷ Base Case 2.
return 0
if palinS[$start$][end] $\neq$ $null$ **then** ▷ Check if already solved
return palinS[start][end]
if inputString[$start$] ==inputString[end] **then** ▷ Solve recursively.
palinS[$start$][end] $\leftarrow$ 1+PALINR($start+1, end-1, inputString, palinS$)
else
palinS[$start$][end] $\leftarrow$ max(PALINR($start+1, end, inputString, palinS$), PALINR($start, end-1], inputString, palinS$))
return palinS[$start$][end]

5. Each entry of the array takes constant work, and we compute values for about half of the $n(n+1)$ entries, which gives us an $O(n^2)$ algorithm in total.

2. St. Thomas Cafeteria Food Inc. is considering opening a series of restaurants along Summit Avenue. There are n viable locations along this road, and the positions of these locations relative to the start of the road are, in miles and in increasing order, given in an array $M = m_1, m_2, \ldots, m_n$. The constraints are as follows:

- At each location, St. Thomas Cafeteria Food Inc. may open at most one restaurant. The expected profit from opening a restaurant at location i is p_i, where $p_i > 0$ and $i \in 1, 2, \ldots, n$.
- Any two restaurants must be at least k miles apart, where k is a positive integer.

Give an efficient dynamic programming algorithm to compute the maximum expected total profit subject to the given constraints.

Solution: In this problem, we just need to compute the maximum profit (not return the list of stores to open), so our algorithm will not worry about storing the best sequence of restaurants to open.

1. Let $BestProfit(i)$ be the most profit you can make among the first i possible restaurant openings.

2. Say you open the ith restaurant. Then you can't open any restaurants between $m_i - k$ and m_i, but there are no restrictions on restaurants before hand. For every i, let m_{C_i} be the location of the last possible restaurant before $m_i - k$ (in other words, $C_i = \max_j |m_j < m_i - k.$). Then in this case, $BestProfit(i) = p_i + BestProfit(C_i)$. If you do not open restaurant i, then $BestProfit(i) = BestProfit(i-1)$. Again, you choose the better of these two options, $BestProfit(i) = \max(BestProfit(i-1), p_i + BestProfit([\max_j |m_j < m_i - k]))$.

3. At each step, we need to find $\max_j |m_j < m_i - k$. We could do this using a binary search, but this adds $O(\log(n))$ work to each step, and would result in an $O(n \log n)$ algorithm. Instead, we will precompute ALL the values of C_i for each i in $O(n)$ total time before hand, store it in an array C, then use this value in the algorithm. This

will result in an $O(n)$ total algorithm - $O(n)$ for precomputation and $O(n)$ for the dynamic program.

(We could have also computed all the values of C on the fly inside the main recursive loop, but this will be easier to follow).

We'll set $C[i] = 0$ to mean that no restaurant is available k miles behind i (in other words, i is less than k miles in front of the first restaurant)

function ComputeBestProfit(Array $m[1 \dots n]$, Array $p[1 \dots n]$, k)
  Array $C[0 \dots n]$ ▷ $C[i]$ will store the closest rest. k miles behind rest. i
  $C[0] = 0$
  for $i = 1$ to n **do**
    $C[i] \leftarrow C[i-1]$ ▷ Start at the previous value
    while $m[C[i]] < m[i] - k$ **do**
      $C[i] \leftarrow C[i] + 1$
    $C[i] \leftarrow C[i] - 1$ ▷ We will increment once too many times
    ▷ After this, $C[n]$ has been filled
  Array $BP[0 \dots n]$ ▷ BP[i] will store BestProfit(i)
  $BP[0] \leftarrow 0$
  for $i = 1$ to n **do**
    $BP[i] = \max BP[i-1], BP[C[i]] + p[i]$
  return BP[n]

4. We have two for loops that each do $O(n)$ work, which means we do a total of $O(n)$ work.

   Note: The reason that the precomputation step only does $O(n)$ work is that the value of $C[i]$ in the while loop only increases from 1 to n, although sporadically. Since it only increases, it can only do so at most n times.

3. **(PRACTICE)** A certain wood-processing plant has a machine that can cut a piece of wood into two pieces at any position. This operation incurs a cost proportional to the length of wood being cut, n, regardless of the location of the cut. Suppose, now, that you want to break the wood into many pieces; you're given the locations of all the cuts you need to make in some array M. The order in which the breaks are made can affect the total cost. For example, if you want to cut a 20-foot piece at positions 3 and 10, then making the first cut at position 3 incurs a total cost of 20+17 = 37, while doing position 10 first has a better cost of 20+10=30.

   Give a dynamic programming algorithm that, given the locations of m cuts in a piece of wood of length n, finds the minimum cost of breaking it into $m + 1$ pieces.

Solution:

1. The general setup is similar to that of chain-matrix multiplication. Lets say the locations of the m cuts were given to you in an array l.

   Let $CutCost(i, j)$ be the minimum cost of making the $j - i - 1$ remaining cuts on the peice of wood between cuts i and j, as if you were given that as one solid piece

with the ends cut off. For notational convenience, we can treat positions $l[0] = 0$ and $l[m+1] = n$ like cuts in our array, so that our end goal is $CutCost(0, m+1)$.

2. Given a segment between cuts i and j, we get to choose which cut to make next, and we incur a total cost of $l[j] - l[i]$ no matter where we make the cut. What we will be left with is two peices of wood that we will then have to cut further. We want to make the choice that creates the least additional cost out of all the possible locations of the next cut.

   Thus, $CutCost(i, j) = l[j] - l[i] + \min_{i<x<j}(CutCost(i, x) + CutCost(x, j))$. Note that to solve $CutCost(i, j)$, we need to know values of $CutCost$ solely on subproblems with a smaller difference between the indices. Thus, in our for loop, we will solve all subproblems of a particular distance between indices before increasing. Also, the base cases can be $CutCost[i, i+1] = 0$, since that piece of wood no longer needs to be cut.

3. **function** CutCost(Array $l[0 \dots m+1]$)
   Array $CC[0 \dots m+1 \times 0 \dots m+1]$
   for $i = 0$ to m **do**
   $CC[i, i+1] \leftarrow 0$
   ▷ Initialization of values
   for $l = 2$ to $m+1$ **do** ▷ The distance between i and j
   for $i = 0$ to $m+1-l$ **do**
   $CC[i, i+l] \leftarrow l[j] - l[i] + \min_{i<x<j}(CC[i, x] + CC[x, j])$
   return $CC[0, m]$

4. We have $O(n^2)$ entries in our array and do $O(n)$ work to solve each entry, resulting in an $O(n^3)$ algorithm.