2. Dynamic Programming

This problem will walk you through the steps of designing a dynamic program. The problem we are solving is the longest palindrome problem: given a string, S, what is the length of the longest palindrome that is a substring of S? A palindrome is a string that reads the same forwards as backwards. For example, "racecar", "eve", and "I", are all palindromes. We also consider the empty string to be a palindrome (of length 0).

(a) First we need to figure out what our subproblems are. Since we are working with strings, a natural subproblem to use is substrings. Let $OPT(i, n)$ denote the length of the longest palindrome in the substring of length n starting at index i. Write an expression for the recursive case of $OPT(i, n)$. (Hint: All palindromes above a certain size have palindromes as substrings).

$$OPT(i,n) = \begin{cases} 2 + OPT(i+1, n-2) & \text{if } S[i] = S[i+n-1] \text{ And } OPT(i+1, n-2) = n-2 \\ MAX(OPT(i, n-1), OPT(i+1, n-1)) & \text{otherwise} \end{cases}$$

(b) Next we need a base case for our OPT recurrence. Write an expression for the base case(s) of this recurrence. (Hint: Which size strings are always palindromes?)

$OPT(i, 0) = 0$ $\quad n = 0$ for all i

$OPT(i, 1) = 1$ $\quad n = 1$ for all i

(c) Now that we have a complete recurrence, we need to figure out which order to solve the subproblems in. Which subproblems does the recursive case $OPT(i, n)$ require to be calculated before it can be solved?

$OPT(i+1, n-2)$, $OPT(i, n-1)$, $OPT(i+1, n-1)$

(d) Given these dependencies, what order should we loop over the subproblem in?

Using the subproblems for strings of shorter length "n" and the first index of substring "i" is the key. We can loop over these subproblems in order of increasing the consecutive index, keep the longest length.

(e) We have all of the pieces required to put together a dynamic program now. Write psuedocode for the dynamic program that computes the length of the longest palindromic substring of S.

(next page)

Initialize OPT [S.length] [S.length + 1] with 0s for all [i][0]
and 1s for all OPT [i] [1]

```
for n from 2 to S.length :
    for i from 0 to S.length - n :
        if (S[i] == S[i+n-1] && OPT [i+1][n-2] = n-2)
            OPT [i] [n] = 2 + OPT [n+1] [n-2]

    OPT[i][n] = MAX (OPT[i+1][n-1],
                     OPT [i] [n-1],
                     OPT [i] [n] )

return OPT [0] [S.length]
```