1. Define matrix[i][j] to be the maximum score that can be gotten by playing first on that [i][j] element. To calculate the other elements we use the recurrence relation matrix[i][j] = max(C[i] + min(matrix[i+2][j], matrix[i+1][j-1]), C[j] + min(matrix[i+1][j-1], matrix[i][j-2])). A) subproblems are the other elements in the array 1 <= i <= n and 1 <= j <= n. B) n^2 subproblems that take O(1) time to store. So time complexity is Θ(n^2). C) results are stored in a 2D array. D) each matrix[i][j] depends on matrix[i+2][j], matrix[i+1][j-1], matrix[i+1][j-1], and matrix[i][j-2]. E) increasing from the top left down to the bottom right and then moving towards the top right. F) for i = 1 to n: for j = 1 to n: compute and store the result of the recurrence function
2. Make a recurrence function that takes the array of candies, the length of the array, current index, your current score, and your current candy as parameters. The base case is that the index is equal to the length of the array. In that case we’ll return your current score. If that isn’t the case check if the current index element of the array is equal to the current candy. If it is equal then recurse the function using the array, the length of the array, the index + 1, current score + 1, and the current candy. If the current candy does not equal the current index element of the array return the max of either the result of recursing the function with the array, the length of the array, the current index + 1, current score – 1, and the type of candy from the array. Or the result of recursing the function with the array, the length of the array, the current index + 1, the current score, and the current candy. A) all of the possible base cases. B) n3 running time. C) one result no memoization. D) every subproblem depends on the other calls to the function that come after it. E) forward. F) function(int arr[], int length, int index, int score, int candy):if (i == n) : return score elseif (arr[index] == candy): return function(arr, length, index + 1, score + 1, candy) else: return max(function(arr, length, index + 1, score – 1, arr[i]), function(arr, length, index + 1, score, candy))
3. Matrix[i][j] is the length of the longest palindromic substring from index i to j. In the matrix anytime i == j the length of the subsequence is 1 because there is only one option. If A[i] == A[j] and j – i = 1 then matrix[i][j] = 2 because it’s a two character string that is a palindrome because the two characters are the same. If A[j] == A[i] and j – i != 1 then matrix[i][j] = matrix[i+1][j-1] + 2. Other than those three options matrix[i][j] = max(matrix[i][j-1], matrix[i+1][j]). A) the subproblems are all of the combinations of i and j in the function. B) O(n2) space and running time because it loops through i and j and takes O(1) to store the result. C) 2D array. D) smaller sections between bigger possible palindromes. E) from the top left to the bottom right to the top right. F) for i = 0 -> n: matrix[i][i] = 1. For end = 2 -> n: for i = 0-> end +1 : j = i + end -1 and compute matrix[i][j]