csce350 — Data Structures and Algorithms
Fall 2020 — Project 4: Hannah and Otto

Assigned: November 16
Due: December 1, 11:55pm
For this assignment only, late submissions will be accepted without penalty until 11:59pm on December 8.

The purpose of this assignment is to give you some practice designing and implementing an algorithm based on the dynamic programming algorithm design strategy.

The Problem Hannah and her friend Otto like palindromes.[1] In fact, they like palindromes so much that, when they see a word that is not a palindrome, they each wonder about how to *insert* letters into that word to make it into a palindrome. They are willing to insert characters that the beginning, the end, or the middle of any word. That's why Hannah and Otto want a program that will read a list of words and *compute the shortest palindrome* that can be formed by inserting letters into each word. Here are some examples.

Word	Letters to insert	Resulting palindrome
roger	E, G	rEGoger
mimi	I	Imimi
benny	E, B, Y	YbennEBy

Your task Your job is to write a program that uses dynamic programming to solve this problem. Specifically, you should write a C++ program that does precisely these things:

1. Read a word, consisting of a sequence of lowercase letters, from standard input.
2. Compute the length of the shortest palindrome that can be formed by inserting characters into that word. Compute the palindrome itself, using uppercase for the inserted letters. If there is more than one shortest palindrome—which happens often—compute the one that is alphabetically first.[2]
3. Print a line containing the length and the palindrome itself, separated by a space.
4. Return to Step 1 above to read and process another instance. Terminate when standard input reaches end-of-file.

Here are a few example inputs with the correct output for each one.

Sample Input 1	***Sample Output 1***
`leonardo`	`13 leoDRAnardoEL`
`michelangelo`	`21 OLmicheGNAlangeHCIMlo`
`donatello`	`14 donatellETANoD`
`raphael`	`11 LEraHphaRel`
`splinter`	`15 RETNILPsplinter`
`april`	`9 LIRPapril`
`casey`	`9 YESAcasey`
`shredder`	`10 shredderHS`
`kraang`	`10 GNkraaRKng`

An additional much longer sample input, along with its correct corresponding output, is available on the course website.

[1] You will probably recall that a palindrome is a string that reads the same both forward and backward.
[2] Specifically, use the ASCII ordering, in which capital letters come before lowercase letters.

Some hints

1. Focus first on finding the *length* of the shortest palindrome, rather on computing the shortest palindrome itself. When you get this part, not only will you have already earned substantial partial credit, but you should also have a much better idea about how to compute the palindromes themselves.

2. A correct dynamic programming solution to this problem should take $\Theta(n^2)$ time. For comparison, my solution takes about 1.5 seconds on my laptop to solve the 62,886 instances in `project4-2.in`. Though the specific speeds are not too important, if your program takes more than a few seconds on this input, something is wrong.

3. Because your solution should use dynamic programming, the first step is to write a recurrence that describes the solution you want to compute, in terms of the solutions to smaller subproblems. Since this is a string problem with a single input $S[0, \ldots, n-1]$, one natural place to start is to think about all of the *substrings* of S. To define a substring, we need to know the position of its first character and its last character, which we'll call i and j respectively. Based on that idea, we can define $p(i, j)$ this way:

Let $p(i, j)$ denote the length of the shortest palindrome that can be formed by inserting into $S[i, \ldots, j]$.

   Then to solve our original problem, we simply need to compute $p(0, n-1)$.

4. Next, we need to figure out how the p values are related to each other. That is: Given values for i and j, can we write an equation for $p(i, j)$ in terms of other p values?

   In this case we are interested in the shortest palindrome that can be formed from $S[i, \ldots, j]$. Let's call that string $X_{i,j}$. We know, from the definition of the word palindrome, that the first and last letters of $X_{i,j}$ are the same—otherwise, $X_{i,j}$ would not be a palindrome. We also know that every character in $X_{i,j}$ comes from one of two places: It could be a character *originally* from S (these are shown in lowercase in our examples) or it could have been *inserted* to complete the palindrome (which we show in uppercase). These two possibilities, applied to the first and last characters of $X_{i,j}$, give us four cases.

case	first character of X_{ij}	last character of X_{ij}
1	original	original
2	original	inserted
3	inserted	original
4	inserted	inserted

   Let's think about each of these cases, in order.

   - Case 1: This case refers to the situation in which both the first and last characters of X_{ij} are originals. If that's true, then we know that $X_{i,j}$ looks like this:

     $$X_{i,j} = S[i]X_{i+1,j-1}S[j]$$

     This is useful, because it begins to connect the solution for (i, j) to the solution to a smaller subproblem, specifically to $(i+1, j-1)$. However, *this case only applies when $S[i] = S[j]$*, because otherwise we would not be getting a palindrome.
   - Case 2: Next, suppose that the first character of $X_{i,j}$ is an original, and the last one was inserted to match it. If that's true, then we know that $X_{i,j}$ looks like this:

     $$X_{i,j} = S[i]X_{i+1,j}S[i].$$

     Again, this gives us a connection to a smaller subproblem.

- Case 3: The *last* character of $X_{i,j}$ is an original, and the *first* one was inserted to match it. This is just like Case 1, but in reverse:
$$X_{i,j} = S[j]X_{i,j-1}S[j]$$
- Case 4: Both the first and last characters of X_{ij} were inserted. This case never happens in shortest palindromes, because if it did, we could get a shorter palindrome by deleting the first and last characters. Therefore, we can completely ignore this case.

That leaves three possibilities. How do we know which one forms the correct palindrome? Since we want the shortest palindrome, the answer is to *try each one, and keep the one that leads to the shortest string*.

All of this is enough to write our recurrence for $p(i, j)$. We take the general forms for X_{ij} from above, and add up the length of each of those strings;

$$p(i,j) = \begin{cases} \min\{p(i+1,j-1)+2, p(i+1,j)+2, p(i,j-1)+2\} & \text{if } S[i] = S[j] \\ \min\{p(i+1,j)+2, p(i,j-1)+2\} & \text{if } S[i] \neq S[j] \end{cases}.$$

The $+2$ terms come from that fact that we are adding two additional characters—one at the front and one at the back—to the smaller string (either $X_{i+1,j-1}$, $X_{i+1,j}$, or $X_{i,j-1}$) that we expand to get X_{ij}. It turns out, though, that if Case 1 applies, that it always will be the shortest length. That allows us to simplify the recurrence a bit, by leaving Cases 2 and 3 out of the top line:

$$\boxed{p(i,j) = \begin{cases} p(i+1,j-1)+2 & \text{if } S[i] = S[j] \\ \min\{p(i+1,j)+2, p(i,j-1)+2\} & \text{if } S[i] \neq S[j] \end{cases}}$$

This simplified version has a box around it because *that formula is the most important thing in this document*—a correct solution to Project 3 will come from turning that recurrence into an implemented algorithm.

5. Now that we have a recurrence, there are a few issues left to resolve.
   (a) We need two loops to fill in the dynamic programming table, one for i and one for j. What are the correct upper and lower bounds for each loop? Should they be increasing or decreasing? What size should the table be?
   (b) What is the base case (or base cases) for p?
   (c) Given the completed DP table, how can you compute the correct shortest palindrome? (Alternatively, can you build the correct palindrome along the way?)

   These details are left for you to resolve, but feel free to ask for help.

Notes A few possibly helpful comments:

- For calibration purposes, my solution has 66 lines of code. Yours may be longer, but if you find yourself writing huge amounts of code, something has probably gone wrong.
- Keep in mind that you are expected to submit your own original work for this problem. Accessing reference material for C++ in general is fine and encouraged. However, you are strongly discouraged from conducting web searches for code specific to palindromes.

What to Submit You should submit, using the department's dropbox website, a single C++ source file containing all of the code for your program. We will compile this program using this command line:

```
g++ -Wall -std=c++11 yourfile.cpp
```