Given a string, find the minimum number of characters to be inserted to convert it to palindrome.

Before we go further, let us understand with few examples:
ab: Number of insertions required is 1. **b**ab
aa: Number of insertions required is 0. aa
abcd: Number of insertions required is 3. **dcb**abcd
abcda: Number of insertions required is 2. a**dc**bcda which is same as number of insertions in the substring bcd(Why?).
abcde: Number of insertions required is 4. **edcb**abcde

Let the input string be *str[l……h]*. The problem can be broken down into three parts:
1. Find the minimum number of insertions in the substring str[l+1,…….h].
2. Find the minimum number of insertions in the substring str[l…….h-1].
3. Find the minimum number of insertions in the substring str[l+1……h-1].

Recursive Solution
The minimum number of insertions in the string str[l…..h] can be given as:
minInsertions(str[l+1…..h-1]) if str[l] is equal to str[h]
min(minInsertions(str[l…..h-1]), minInsertions(str[l+1…..h])) + 1 otherwise

```
// A Naive recursive program to find minimum number insertions
// needed to make a string palindrome
#include <stdio.h>
#include <limits.h>
#include <string.h>

// A utility function to find minimum of two numbers
int min(int a, int b)
{  return a < b ? a : b; }

// Recursive function to find minimum number of insersions
int findMinInsertions(char str[], int l, int h)
{
    // Base Cases
    if (l > h) return INT_MAX;
    if (l == h) return 0;
    if (l == h - 1) return (str[l] == str[h])? 0 : 1;

    // Check if the first and last characters are same. On the basis of the
    // comparison result, decide which subrpoblem(s) to call
    return (str[l] == str[h])? findMinInsertions(str, l + 1, h - 1):
                               (min(findMinInsertions(str, l, h - 1),
                                    findMinInsertions(str, l + 1, h)) + 1);
}

// Driver program to test above functions
int main()
{
    char str[] = "geeks";
    printf("%d", findMinInsertions(str, 0, strlen(str)-1));
    return 0;
}
```

Output:

```
3
```

Dynamic Programming based Solution
If we observe the above approach carefully, we can find that it exhibits overlapping subproblems.
Suppose we want to find the minimum number of insertions in string “abcde”:

```
                      abcde
                 /      |      \
                /       |       \
            bcde        abcd      bcd  <- case 3 is discarded as str[l] != str[h]
          /   |   \      /   |   \
         /    |    \    /    |    \
       cde   bcd  cd   bcd abc bc
      / | \  / | \ /|\ / | \
     de cd d cd bc c……….
```

The substrings in bold show that the recursion to be terminated and the recursion tree cannot originate from there. Substring in the same color indicates overlapping subproblems.

How to reuse solutions of subproblems?
We can create a table to store results of subproblems so that they can be used directly if same subproblem is encountered again.

The below table represents the stored values for the string abcde.

```
a b c d e
----------
0 1 2 3 4
0 0 1 2 3
0 0 0 1 2
0 0 0 0 1
0 0 0 0 0
```

How to fill the table?
The table should be filled in diagonal fashion. For the string abcde, 0….4, the following should be order in which the table is filled:

```
Gap = 1:
(0, 1) (1, 2) (2, 3) (3, 4)

Gap = 2:
(0, 2) (1, 3) (2, 4)

Gap = 3:
(0, 3) (1, 4)

Gap = 4:
(0, 4)
```

```
// A Dynamic Programming based program to find minimum number
// insertions needed to make a string palindrome
#include <stdio.h>
#include <string.h>

// A utility function to find minimum of two integers
int min(int a, int b)
{   return a < b ? a : b;  }

// A DP function to find minimum number of insersions
int findMinInsertionsDP(char str[], int n)
{
    // Create a table of size n*n. table[i][j] will store
    // minumum number of insertions needed to convert str[i..j]
```

```
    // to a palindrome.
    int table[n][n], l, h, gap;

    // Initialize all table entries as 0
    memset(table, 0, sizeof(table));

    // Fill the table
    for (gap = 1; gap < n; ++gap)
        for (l = 0, h = gap; h < n; ++l, ++h)
            table[l][h] = (str[l] == str[h])? table[l+1][h-1] :
                          (min(table[l][h-1], table[l+1][h]) + 1);

    // Return minimum number of insertions for str[0..n-1]
    return table[0][n-1];
}

// Driver program to test above function.
int main()
{
    char str[] = "geeks";
    printf("%d", findMinInsertionsDP(str, strlen(str)));
    return 0;
}
```

Output:

```
3
```

Time complexity: O(N^2)
Auxiliary Space: O(N^2)

Another Dynamic Programming Solution (Variation of Longest Common Subsequence Problem)
The problem of finding minimum insertions can also be solved using Longest Common Subsequence (LCS) Problem. If we find out LCS of string and its reverse, we know how many maximum characters can form a palindrome. We need insert remaining characters. Following are the steps.
1) Find the length of LCS of input string and its reverse. Let the length be 'l'.
2) The minimum number insertions needed is length of input string minus 'l'.

```
// An LCS based program to find minimum number insertions needed to
// make a string palindrome
#include<stdio.h>
#include <string.h>

/* Utility function to get max of 2 integers */
int max(int a, int b)
{   return (a > b)? a : b; }

/* Returns length of LCS for X[0..m-1], Y[0..n-1].
   See http://goo.gl/bHQVP for details of this function */
int lcs( char *X, char *Y, int m, int n )
{
   int L[n+1][n+1];
   int i, j;

   /* Following steps build L[m+1][n+1] in bottom up fashion. Note
      that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] */
   for (i=0; i<=m; i++)
   {
     for (j=0; j<=n; j++)
     {
       if (i == 0 || j == 0)
         L[i][j] = 0;

       else if (X[i-1] == Y[j-1])
         L[i][j] = L[i-1][j-1] + 1;
```

```
        else
          L[i][j] = max(L[i-1][j], L[i][j-1]);
      }
    }

    /* L[m][n] contains length of LCS for X[0..n-1] and Y[0..m-1] */
    return L[m][n];
}

// LCS based function to find minimum number of insersions
int findMinInsertionsLCS(char str[], int n)
{
    // Creata another string to store reverse of 'str'
    char rev[n+1];
    strcpy(rev, str);
    strrev(rev);

    // The output is length of string minus length of lcs of
    // str and it reverse
    return (n - lcs(str, rev, n, n));
}

// Driver program to test above functions
int main()
{
    char str[] = "geeks";
    printf("%d", findMinInsertionsLCS(str, strlen(str)));
    return 0;
}
```

Output:

```
3
```

Time complexity of this method is also O(n^2) and this method also requires O(n^2) extra space.