



# Palindromic Partitioning

#### We'll cover the following

^

- Problem Statement
- Basic Solution
- Top-down Dynamic Programming with Memoization
- Bottom-up Dynamic Programming

## Problem Statement #

Given a string, we want to cut it into pieces such that each piece is a palindrome. Write a function to return the minimum number of cuts needed.

## Example 1:

```
Input: "abdbca"
Output: 3
Explanation: Palindrome pieces are "a", "bdb", "c", "a".
```

# Example 2:

```
Input: = "cddpd"
Output: 2
Explanation: Palindrome pieces are "c", "d", "dpd".
```

# Example 3:

```
Input: = "pqr"
Output: 2
Explanation: Palindrome pieces are "p", "q", "r".
```

## Example 4:

```
Input: = "pp"
Output: 0
Explanation: We do not need to cut, as "pp" is a palindrome.
```

## **Basic Solution #**

This problem follows the Longest Palindromic Subsequence (https://www.educative.io/collection/page/5668639101419520/5633779737559040/574811928317 1328/) pattern and shares a similar approach as that of the Longest Palindromic Substring

(https://www.educative.io/collection/page/5668639101419520/5633779737559040/5661601461960704/).

The brute-force solution will be to try all the substring combinations of the given string. We can start processing from the beginning of the string and keep adding one character at a time. At any step, if we get a palindrome, we take it as one piece and recursively process the remaining length of the string to find the minimum cuts needed.

Here is the code:

```
(§) JS
                                     © C++
👙 Java
                       Python3
 1 def find_MPP_cuts(st):
      return find_MPP_cuts_recursive(st, 0, len(st)-1)
 2
 3
 4
 5 def find_MPP_cuts_recursive(st, startIndex, endIndex):
      # we don't need to cut the string if it is a palindrome
 6
 7
      if startIndex >= endIndex or is_palindrome(st, startIndex, endIndex):
 8
        return 0
 9
10
      # at max, we need to cut the string into its 'length-1' pieces
11
      minimumCuts = endIndex - startIndex
12
      for i in range(startIndex, endIndex+1):
13
        if is_palindrome(st, startIndex, i):
14
          # we can cut here as we have a palindrome from 'startIndex' to 'i'
15
          minimumCuts = min(
            minimumCuts, 1 + find_MPP_cuts_recursive(st, i + 1, endIndex))
16
17
18
      return minimumCuts
19
20
21 def is_palindrome(st, x, y):
22
      while (x < y):
23
        if st[x] != st[y]:
24
          return False
25
        x += 1
26
        y -= 1
27
      return True
28
29
30 def main():
31
      print(find_MPP_cuts("abdbca"))
32
      print(find_MPP_cuts("cdpdd"))
      print(find_MPP_cuts("pqr"))
33
34
      print(find_MPP_cuts("pp"))
35
      print(find_MPP_cuts("madam"))
36
37
38 main()
\triangleright
                                                                                             []
```

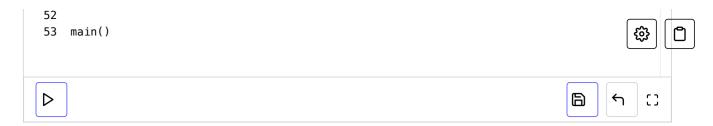
The time complexity of the above algorithm is exponential  $O(2^n)$ , where 'n' is the length of the input string. The space complexity is O(n) which is used to store the recursion stack.

# Top-down Dynamic Programming with Memoization #

We can memoize both functions findMPPCutsRecursive() and isPalindrome(). The two changing values in both these functions are the two indexes; therefore, we can store the results of all the subproblems in a two-dimensional array. (alternatively, we can use a hash-table).

Here is the code:

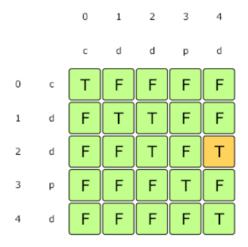
```
👙 Java
           (S) JS
                       Python3
                                     G C++
 1 def find_MPP_cuts(st):
      n = len(st)
      dp = [[-1 for _ in range(n)] for _ in range(n)]
 3
 4
      dpIsPalindrome = [[-1 for _ in range(n)] for _ in range(n)]
 5
      return find_MPP_cuts_recursive(dp, dpIsPalindrome, st, 0, n - 1)
 6
 7
   def find_MPP_cuts_recursive(dp, dpIsPalindrome, st, startIndex, endIndex):
 8
 9
      if startIndex >= endIndex or is_palindrome(dpIsPalindrome, st, startIndex, endIndex):
10
11
        return 0
12
13
      if dp[startIndex][endIndex] == -1:
14
        # at max, we need to cut the string into its 'length-1' pieces
15
        minimumCuts = endIndex - startIndex
16
        for i in range(startIndex, endIndex+1):
17
          if is_palindrome(dpIsPalindrome, st, startIndex, i):
            # we can cut here as we have a palindrome from 'startIndex' to 'i'
18
19
            minimumCuts = min(
20
              minimumCuts, 1 + find_MPP_cuts_recursive(dp, dpIsPalindrome, st, i + 1, endIndex
21
        dp[startIndex][endIndex] = minimumCuts
22
23
24
      return dp[startIndex][endIndex]
25
26
   def is_palindrome(dpIsPalindrome, st, x, y):
27
28
      if dpIsPalindrome[x][y] == -1:
29
        dpIsPalindrome[x][y] = 1
30
        i, j = x, y
31
        while i < j:
32
          if st[i] != st[j]:
33
            dpIsPalindrome[x][y] = 0
34
            break
35
          i += 1
          j -= 1
36
37
          # use memoization to find if the remaining string is a palindrome
          if i < j and dpIsPalindrome[i][j] != -1:
38
39
            dpIsPalindrome[x][y] = dpIsPalindrome[i][j]
40
            break
41
42
      return True if dpIsPalindrome[x][y] == 1 else False
43
44
45 def main():
      print(find_MPP_cuts("abdbca"))
46
47
      print(find_MPP_cuts("cdpdd"))
48
      print(find_MPP_cuts("pqr"))
49
      print(find_MPP_cuts("pp"))
50
      print(find_MPP_cuts("madam"))
51
```



# **Bottom-up Dynamic Programming #**

The above solution tells us that we need to build two tables, one for the <code>isPalindrome()</code> and one for finding the minimum cuts needed.

If you remember, we built a table in the Longest Palindromic Substring (https://www.educative.io/collection/page/5668639101419520/5633779737559040/566160146196 0704/) (LPS) chapter that can tell us what substrings (of the input string) are palindrome. We will use the same approach here to build the table required for <code>isPalindrome()</code>. For example, here is the final output from LPS for "cddpd". From this table we can clearly see that the <code>substring(2,4) => 'dpd'</code> is a palindrome:



To build the second table for finding the minimum cuts, we can iterate through the first table built for <code>isPalindrome()</code>. At any step, if we get a palindrome, we can cut the string there. Which means minimum cuts will be one plus the cuts needed for the remaining string.

Here is the code for the bottom-up approach:

```
👙 Java
           (§) JS
                       Python3
                                     G C++
    def find_MPP_cuts(st):
 2
      n = len(st)
      # isPalindrome[i][j] will be 'true' if the string from index 'i' to index 'j' is a palin
 3
      isPalindrome = [[False for _ in range(n)] for _ in range(n)]
 4
 5
 6
      # every string with one character is a palindrome
 7
      for i in range(n):
        isPalindrome[i][i] = True
 8
 9
10
      # populate isPalindrome table
11
      for startIndex in range(n-1, -1, -1):
12
        for endIndex in range(startIndex+1, n):
          if st[startIndex] == st[endIndex]:
13
```

```
# if it's a two character string or if the remaining string is a palindrome
14
15
            if endIndex - startIndex == 1 or isPalindrome[startIndex + 1][endIndex - 1] to isPalindrome[startIndex + 1][endIndex - 1]
               isPalindrome[startIndex][endIndex] = True
16
17
18
      # now lets populate the second table, every index in 'cuts' stores the minimum cuts need
19
      # for the substring from that index till the end
20
      cuts = [0 for _ in range(n)]
21
      for startIndex in range(n-1, -1, -1):
        minCuts = n # maximum cuts
22
23
        for endIndex in range(n-1, startIndex-1, -1):
          if isPalindrome[startIndex][endIndex]:
24
            # we can cut here as we got a palindrome
25
            # also we don't need any cut if the whole substring is a palindrome
26
27
            minCuts = 0 if endIndex == n-1 else min(minCuts, 1 + cuts[endIndex + 1])
28
29
        cuts[startIndex] = minCuts
30
      return cuts[0]
31
32
33
    def main():
34
35
      print(find_MPP_cuts("abdbca"))
36
      print(find_MPP_cuts("cdpdd"))
37
      print(find_MPP_cuts("pqr"))
38
      print(find_MPP_cuts("pp"))
39
      print(find_MPP_cuts("madam"))
40
41
    main()
42
D
                                                                                               :3
```

The time and space complexity of the above algorithm is  $O(n^2)$ , where 'n' is the length of the input string.

