**Palindromic Partitioning**

**WE'LL COVER THE FOLLOWING**

* + - [Problem Statement](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxxqrE2kKrY#problem-statement)
    - [Basic Solution](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxxqrE2kKrY#basic-solution)
    - [Top-down Dynamic Programming with Memoization](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxxqrE2kKrY#top-down-dynamic-programming-with-memoization)
    - [Bottom-up Dynamic Programming](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxxqrE2kKrY#bottom-up-dynamic-programming)

**Problem Statement**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxxqrE2kKrY#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**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxxqrE2kKrY#basic-solution)

This problem follows the [Longest Palindromic Subsequence](https://www.educative.io/collection/page/5668639101419520/5633779737559040/5748119283171328/) 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:

* [Java](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxxqrE2kKrY)
* [JS](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxxqrE2kKrY)
* [Python3](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxxqrE2kKrY)
* [C++](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxxqrE2kKrY)

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

    if(startIndex >= endIndex || isPalindrome(st, startIndex, endIndex))

      return 0;

    // at max, we need to cut the string into its 'length-1' pieces

    int minimumCuts = endIndex-startIndex;

    for (int i=startIndex; i <= endIndex; i++) {

      if(isPalindrome(st, startIndex, i)){

        // we can cut here as we have a palindrome from 'startIndex' to 'i'

        minimumCuts = Math.min(minimumCuts, 1 + findMPPCutsRecursive(st, i+1, endIndex));

      }

    }

    return minimumCuts;

  }

  private boolean isPalindrome(String st, int x, int y) {

    while(x < y) {

      if(st.charAt(x++) != st.charAt(y--))

        return false;

    }

    return true;

  }

  public static void main(String[] args) {

    MPP mpp = new MPP();

    System.out.println(mpp.findMPPCuts("abdbca"));

    System.out.println(mpp.findMPPCuts("cdpdd"));

    System.out.println(mpp.findMPPCuts("pqr"));

    System.out.println(mpp.findMPPCuts("pp"));

   }

}





RUN

SAVERESET

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

**Top-down Dynamic Programming with Memoization**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxxqrE2kKrY#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](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxxqrE2kKrY)
* [JS](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxxqrE2kKrY)
* [Python3](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxxqrE2kKrY)
* [C++](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxxqrE2kKrY)

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

    }

    return dp[startIndex][endIndex];

  }

  private boolean isPalindrome(Boolean dpIsPalindrome[][], String st, int x, int y) {

    if(dpIsPalindrome[x][y] == null) {

      dpIsPalindrome[x][y]=true;

      int i=x, j=y;

      while(i < j) {

        if(st.charAt(i++) != st.charAt(j--)) {

          dpIsPalindrome[x][y]=false;

          break;

        }

        // use memoization to find if the remaining string is a palindrome

        if(i < j && dpIsPalindrome[i][j] != null) {

          dpIsPalindrome[x][y] = dpIsPalindrome[i][j];

          break;

        }

      }

    }

    return dpIsPalindrome[x][y];

  }

  public static void main(String[] args) {

    MPP mpp = new MPP();

    System.out.println(mpp.findMPPCuts("abdbca"));

    System.out.println(mpp.findMPPCuts("cdpdd"));

    System.out.println(mpp.findMPPCuts("pqr"));

    System.out.println(mpp.findMPPCuts("pp"));

   }

}





RUN

SAVERESET

**Bottom-up Dynamic Programming**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxxqrE2kKrY#bottom-up-dynamic-programming)

The above solution tells us that we need to build two tables, one for the isPalindrome() 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/5668639101419520/5633779737559040/5661601461960704/) (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 isPalindrome(). For example, here is the final output from LPS for “cddpd”. From this table we can clearly see that the substring(2,4) => 'dpd' is a palindrome:

To build the second table for finding the minimum cuts, we can iterate through the first table built for isPalindrome(). 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](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxxqrE2kKrY)
* [JS](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxxqrE2kKrY)
* [Python3](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxxqrE2kKrY)
* [C++](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxxqrE2kKrY)

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

      }

    }

    // now lets populate the second table, every index in 'cuts' stores the minimum cuts needed

    // for the substring from that index till the end

    int[] cuts = new int[st.length()];

    for (int startIndex = st.length() - 1; startIndex >= 0; startIndex--) {

      int minCuts = st.length(); // maximum cuts

      for (int endIndex = st.length() - 1; endIndex >= startIndex; endIndex--) {

        if (isPalindrome[startIndex][endIndex]) {

          // we can cut here as we got a palindrome

          // also we dont need any cut if the whole substring is a palindrome

          minCuts = (endIndex == st.length() - 1) ? 0 : Math.min(minCuts, 1 + cuts[endIndex + 1]);

        }

      }

      cuts[startIndex] = minCuts;

    }

    return cuts[0];

  }

  public static void main(String[] args) {

    MPP mpp = new MPP();

    System.out.println(mpp.findMPPCuts("abdbca"));

    System.out.println(mpp.findMPPCuts("cdpdd"));

    System.out.println(mpp.findMPPCuts("pqr"));

    System.out.println(mpp.findMPPCuts("pp"));

    System.out.println(mpp.findMPPCuts("madam"));

  }

}





RUN

SAVERESET

The time and space complexity of the above algorithm is O(n^2)*O*(*n*​2​​), where ‘n’ is the length of the input string.

**MARK AS COMPLETED**

[**←    Back**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gkX4prBkRLj)

Minimum Deletions in a String to make it a Palindrome

[**Next    →**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/RMkk7NwE44R)

Longest Common Substring

Stuck? Get help on

[**DISCUSS**](https://discuss.educative.io/c/grokking-dynamic-programming-patterns-for-coding-interviews-design-gurus/pattern-3-palindromic-subsequence-palindromic-partitioning)

Send feedback

18 Recommendations