# Manacher's Algorithm - GeeksforGeeks

**Source:**
https://www.geeksforgeeks.org/manachers-algorithm-linear-time-longest-palindromic-substring-part-1/

 Manacher's Algorithm Last Updated : 30 Jul, 2025 Manacher’s Algorithm is an algorithm used to find all palindromic substrings of a string in linear time. It is mainly applied to problems involving palindromes, especially when fast processing is needed. It is commonly used to solve: Longest Palindromic Substring - Find the longest contiguous substring that is a palindrome in O(n) time. Multiple Palindrome Queries - After O(n) preprocessing, answer queries like “Is s[i..j] a palindrome?” in O(1) per query. Traditional methods like brute-force or dynamic programming take $O(n^2)$ time and Rabin Karp (Rolling Hash) take $O(n \times \log n)$ time which is inefficient for large strings or multiple queries. Manacher’s Algorithm solves these problems efficiently in O(n) time. Why Brute-Force, DP, Center Expansion or Hashing Are Not Efficient Enough Before diving into Manacher’s Algorithm, let’s briefly look at common approaches used for palindrome-related problems and why they fall short in terms of efficiency. Brute-Force Check every possible substring and test if it's a palindrome by comparing characters. Total substrings = $O(n^2)$, and checking each takes up to O(n) time. Time Complexity: $O(n^3)$ Drawback: Very slow even for moderate input sizes. Dynamic Programming (DP) Use a 2D table where dp[i][j] stores whether s[i..j] is a palindrome. Build the table using previously computed results. Time Complexity: $O(n^2)$ Space Complexity: $O(n^2)$ Drawback: Still too slow and heavy in space for large strings. Center Expansion Treat each character and each gap between characters as a potential center For each center: Expand outward while s[left] == s[right] Track the longest palindrome found during expansions Time Complexity: $O(n^2)$ Space Complexity: O(1) Drawbacks: Still quadratic in time in the worst case, Less efficient than Manacher’s Algorithm for long strings Rabin-Karp (Hashing) + Binary Search Use rolling hash to compare reversed substrings with the original. Combine with binary search to find the longest palindrome centered at each position. Time Complexity: O(n log n) due to binary search on each center. Space Complexity: O(n) for prefix hashes. Drawback: Much faster than DP but still not optimal. Also, care needed to avoid hash collisions. Why Manacher’s is Better: All the above methods are slower than linear time. Manacher’s Algorithm solves the same problems in O(n) time using a clever technique based on symmetry and center expansion, with constant space (excluding the result array). Preprocessing the String Why Preprocessing is Needed ? In a normal string, palindromic substrings can be of odd or even length: Odd-length: "aba", "racecar" Even-length: "abba", "noon" Handling these two cases separately makes the implementation complex. To simplify the logic and treat all palindromes uniformly, Manacher’s Algorithm modifies the original string so that: All palindromes become odd-length Each character and the spaces between characters become potential centers of palindromes This avoids writing separate code for odd and even lengths and allows a single unified expansion logic. How It’s Done in Code: Start with @: Sentinel to handle left boundary and avoid bounds checking during expansion. Insert # between characters => This makes sure that: - Original characters stay separated - Even-length palindromes like "abba" now look like: "#a#b#b#a#" - So, both even and odd cases are treated as odd-length in the transformed string. End with $: Sentinel to handle the right boundary safely. Example: For input string: s = "abba" Transformed string ms: "@#a#b#b#a#$" Each character (including #) can now be treated as a potential center for a palindrome. This transformation simplifies the core logic of the algorithm and ensures consistent behavior. Core Algorithm After preprocessing, the algorithm runs a single loop over the transformed string ms to compute the array p[], where p[i] stores the radius (number of characters on one side) of the longest palindrome centered at position i. This logic is implemented in the runManacher() function. Key Variables: int l = 0, r = 0; l and r represent the left and right boundaries of the rightmost palindrome found so far. Loop Through the Transformed String => Start from index 1 (since index 0 is @, the sentinel) => End before n - 1 (index of $, the ending sentinel) Mirror Optimization: p[i] = max(0, min(r - i, p[r + l - i])); => If i is within the current palindrome [l, r], we use the mirror of i about the center to initialize p[i]. => Mirror position: mirror = l + r - i => We take the minimum between p[mirror] and the remaining span (r - i) to avoid going out of bounds. => This avoids redundant re-computation , a key idea that makes the algorithm linear. Expand Palindrome Centered at i - => Try to expand the palindrome centered at i by comparing characters on both sides. => Continue expanding as long as characters match. C++ while ( ms [ i + 1 + p [ i ]] == ms [ i - 1 - p [ i ]]) { ++ p [ i ]; } Update Rightmost Boundary: If the palindrome centered at i goes beyond the current right boundary r, we update l and r to reflect this new, longer palindrome. C++ if ( i + p [ i ] > r ) { l = i - p [ i ]; r = i + p [ i ]; } Final Result: The array p[i] for each index i in ms contains the maximum radius of palindrome centered at i. Mirror Symmetry & Case-by-Case Connection We now explain how code handles the three mirror symmetry cases: When i < r → i is within current palindrome: int mirror = l + r - i; p[i] = min(r - i, p[mirror]); This mirrors the idea of computing the palindrome at i by referencing its mirror position j = 2*c – i. We are essentially handling Case 1 and Case 2: Case 1: (p[mirror] < r - i): Mirror is fully inside current palindrome → safe to copy full value. Case 2: (p[mirror] > r - i): Mirror extends beyond r, so copy only up to boundary r. p[i] = min(r - i, p[mirror]); which correctly handles both cases using min. When p[mirror] == r - i (Case 3): We still set p[i] = r - i via the min(...) logic above, but... We continue expanding manually with: C++ while ( ms [ i + 1 + p [ i ]] == ms [ i - 1 - p [ i ]]{ ++ p [ i ]; } This is the implementation of Case 3, where the mirrored palindrome just touches the boundary, and you try to expand beyond it. Update of [l, r] interval: Once expansion at i goes past r, we update the rightmost palindrome. This keeps l and r as the bounds of the longest palindrome seen so far, required for correct mirror computations in the next iterations. C++ if ( i + p [ i ] > r ) { l = i - p [ i ]; r = i + p [ i ]; } Why Manacher’s Algorithm Works To understand why Manacher’s Algorithm correctly computes all palindromic substrings in linear time, we need to look at the core idea of symmetry and how it avoids redundant work using mirror positions. Symmetry in Palindromes A palindrome has mirror symmetry around its center. If a palindrome is centered at some position center, and we are at a position i inside that palindrome (i.e., i < r), then the character at i has a mirror position: mirror = 2 * center - i Reusing Information from Mirror If we already know that p[mirror] = x, we can safely say that: At least min(r - i, p[mirror]) characters around i will also match Beyond that, we may or may not be able to expand — so we try This saves time: instead of re-expanding everything from scratch at each center, we start from the known length and only expand beyond it if necessary. C++ #include <iostream> #include <vector> #include <string> using namespace std ; class Manacher { public : // p[i] = radius of longest palindrome centered at i // in transformed string vector < int > p ; // transformed string with # and sentinels string ms ; // preprocess the string and run the algorithm Manacher ( string & s ) { // left sentinel ms = "@" ; for ( char c : s ) { ms += "#" + string ( 1 , c ); } // right sentinel ms += "#$" ; // run Manacher’s algorithm runManacher (); } void runManacher () { int n = ms . size (); p . assign ( n , 0 ); int l = 0 , r = 0 ; for ( int i = 1 ; i < n - 1 ; ++ i ) { // mirror of i around center (l + r)/2 int mirror = l + r - i ; // initialize p[i] based on its mirror // if within bounds if ( i < r ) p [ i ] = min ( r - i , p [ mirror ]); // expand palindrome centered at i while ( ms [ i + 1 + p [ i ]] == ms [ i - 1 - p [ i ]]){ ++ p [ i ]; } // update [l, r] if the palindrome expands // beyond current r if ( i + p [ i ] > r ) { l = i - p [ i ]; r = i + p [ i ]; } } } // returns length of longest palindrome centered // at 'cen' in original string // 'odd' = 1 → check for odd-length, 'odd' = 0 → even-length int getLongest ( int cen , int odd ) { // map original index to transformed string index int pos = 2 * cen + 2 + ! odd ; return p [ pos ]; } // checks if s[l..r] is a palindrome in O(1) bool check ( int l , int r ) { int len = r - l + 1 ; int cen = ( l + r ) / 2 ; return len <= getLongest ( cen , len % 2 ); } }; Java class Manacher { // p[i] = radius of longest palindrome centered at i // in transformed string int [] p ; // transformed string with # and sentinels String ms ; // preprocess the string and run the algorithm Manacher ( String s ) { // left sentinel ms = "@" ; for ( char c : s . toCharArray ()) { ms += "#" + c ; } // right sentinel ms += "#$" ; // run Manacher’s algorithm runManacher (); } void runManacher () { int n = ms . length (); p = new int [ n ] ; int l = 0 , r = 0 ; for ( int i = 1 ; i < n - 1 ; ++ i ) { // mirror of i around center (l + r)/2 int mirror = l + r - i ; // initialize p[i] based on its mirror // if within bounds if ( i < r ) p [ i ] = Math . min ( r - i , p [ mirror ] ); // expand palindrome centered at i while ( ms . charAt ( i + 1 + p [ i ] ) == ms . charAt ( i - 1 - p [ i ] )) { ++ p [ i ] ; } // update [l, r] if the palindrome expands // beyond current r if ( i + p [ i ] > r ) { l = i - p [ i ] ; r = i + p [ i ] ; } } } // returns length of longest palindrome centered // at 'cen' in original string // 'odd' = 1 → check for odd-length, 'odd' = 0 → even-length int getLongest ( int cen , int odd ) { // map original index to transformed string index int pos = 2 * cen + 2 + ( odd == 0 ? 1 : 0 ); return p [ pos ] ; } // checks if s[l..r] is a palindrome in O(1) boolean check ( int l , int r ) { int len = r - l + 1 ; int cen = ( l + r ) / 2 ; return len <= getLongest ( cen , len % 2 ); } } Python class Manacher : # p[i] = radius of longest palindrome centered at i # in transformed string def __init__ ( self , s ): # transformed string with # and sentinels self . ms = "@" for c in s : self . ms += "#" + c self . ms += "#$" # run Manacher's algorithm self . p = [ 0 ] * len ( self . ms ) self . runManacher () def runManacher ( self ): n = len ( self . ms ) l = r = 0 for i in range ( 1 , n - 1 ): # mirror of i around center (l + r)/2 mirror = l + r - i # initialize p[i] based on its mirror # if within bounds if i < r : self . p [ i ] = min ( r - i , self . p [ mirror ]) # expand palindrome centered at i while self . ms [ i + 1 + self . p [ i ]] == \ self . ms [ i - 1 - self . p [ i ]]: self . p [ i ] += 1 # update [l, r] if the palindrome expands # beyond current r if i + self . p [ i ] > r : l = i - self . p [ i ] r = i + self . p [ i ] # returns length of longest palindrome centered # at 'cen' in original string # 'odd' = 1 → check for odd-length, 'odd' = 0 → even-length def getLongest ( self , cen , odd ): # map original index to transformed string index pos = 2 * cen + 2 + ( 0 if odd else 1 ) return self . p [ pos ] # checks if s[l..r] is a palindrome in O(1) def check ( self , l , r ): length = r - l + 1 cen = ( l + r ) // 2 return length <= self . getLongest ( cen , length % 2 ) C# using System ; class Manacher { // p[i] = radius of longest palindrome centered at i // in transformed string public int [] p ; // transformed string with # and sentinels public string ms ; // preprocess the string and run the algorithm public manacher ( string s ) { ms = "@" ; foreach ( char c in s ) { ms += "#" + c ; } ms += "#$" ; runManacher (); } void runManacher () { int n = ms . Length ; p = new int [ n ]; int l = 0 , r = 0 ; for ( int i = 1 ; i < n - 1 ; ++ i ) { // mirror of i around center (l + r)/2 int mirror = l + r - i ; // initialize p[i] based on its mirror // if within bounds if ( i < r ) p [ i ] = Math . Min ( r - i , p [ mirror ]); // expand palindrome centered at i while ( ms [ i + 1 + p [ i ]] == ms [ i - 1 - p [ i ]]) { ++ p [ i ]; } // update [l, r] if the palindrome expands // beyond current r if ( i + p [ i ] > r ) { l = i - p [ i ]; r = i + p [ i ]; } } } // returns length of longest palindrome centered // at 'cen' in original string // 'odd' = 1 → check for odd-length, 'odd' = 0 → even-length public int getLongest ( int cen , int odd ) { int pos = 2 * cen + 2 + ( odd == 0 ? 1 : 0 ); return p [ pos ]; } // checks if s[l..r] is a palindrome in O(1) public bool check ( int l , int r ) { int len = r - l + 1 ; int cen = ( l + r ) / 2 ; return len <= getLongest ( cen , len % 2 ); } } JavaScript class Manacher { constructor ( s ) { // transformed string with # and sentinels this . ms = "@" ; for ( let c of s ) { this . ms += "#" + c ; } this . ms += "#$" ; // p[i] = radius of longest palindrome centered at i // in transformed string this . p = Array ( this . ms . length ). fill ( 0 ); // run Manacher's algorithm this . runManacher (); } runManacher () { let n = this . ms . length ; let l = 0 , r = 0 ; for ( let i = 1 ; i < n - 1 ; ++ i ) { // mirror of i around center (l + r)/2 let mirror = l + r - i ; // initialize p[i] based on its mirror // if within bounds if ( i < r ) this . p [ i ] = Math . min ( r - i , this . p [ mirror ]); // expand palindrome centered at i while ( this . ms [ i + 1 + this . p [ i ]] === this . ms [ i - 1 - this . p [ i ]]) { this . p [ i ] ++ ; } // update [l, r] if the palindrome expands // beyond current r if ( i + this . p [ i ] > r ) { l = i - this . p [ i ]; r = i + this . p [ i ]; } } } // returns length of longest palindrome centered // at 'cen' in original string // 'odd' = 1 → check for odd-length, 'odd' = 0 → even-length getLongest ( cen , odd ) { let pos = 2 * cen + 2 + ( odd === 0 ? 1 : 0 ); return this . p [ pos ]; } // checks if s[l..r] is a palindrome in O(1) check ( l , r ) { let len = r - l + 1 ; let cen = Math . floor (( l + r ) / 2 ); return len <= this . getLongest ( cen , len % 2 ); } } Time Complexity: O(n) Manacher's Algorithm runs in linear time because each character in the transformed string is visited at most once during expansion, and mirror values prevent redundant checks. In total, the algorithm performs O(n) operations where n is the length of the original string (after transformation it's about 2n+3, but still linear). Auxiliary Space: O(n) An extra array p[] of size proportional to the transformed string (i.e., 2n+3) is used to store palindrome radii. The transformed string also takes O(n) space, so the total additional space remains linear in terms of the original string size. Code to Find the Longest Palindromic Substring Using Manacher's C++ class GfG { public : string longestPalindrome ( string s ) { Manacher mob ( s ); int n = s . size (); // maximum length found so far int maxLen = 1 ; // starting index of longest palindrome int bestStart = 0 ; for ( int i = 0 ; i < n ; i ++ ) { // check for odd-length palindrome centered at i int oddLen = mob . getLongest ( i , 1 ); if ( oddLen > maxLen ) { maxLen = oddLen ; bestStart = i - maxLen / 2 ; } // check for even-length palindrome centered // between i and i+1 int evenLen = mob . getLongest ( i , 0 ); if ( evenLen > maxLen ) { maxLen = evenLen ; bestStart = i - maxLen / 2 + 1 ; } } // extract the longest palindromic substring return s . substr ( bestStart , maxLen ); } }; Java class GfG { public String longestPalindrome ( String s ) { Manacher mob = new Manacher ( s ); int n = s . length (); // maximum length found so far int maxLen = 1 ; // starting index of longest palindrome int bestStart = 0 ; for ( int i = 0 ; i < n ; i ++ ) { // check for odd-length palindrome centered at i int oddLen = mob . getLongest ( i , 1 ); if ( oddLen > maxLen ) { maxLen = oddLen ; bestStart = i - maxLen / 2 ; } // check for even-length palindrome centered // between i and i+1 int evenLen = mob . getLongest ( i , 0 ); if ( evenLen > maxLen ) { maxLen = evenLen ; bestStart = i - maxLen / 2 + 1 ; } } // extract the longest palindromic substring return s . substring ( bestStart , bestStart + maxLen ); } } Python class GfG : def longestPalindrome ( self , s ): mob = Manacher ( s ) n = len ( s ) # maximum length found so far maxLen = 1 # starting index of longest palindrome bestStart = 0 for i in range ( n ): # check for odd-length palindrome centered at i oddLen = mob . getLongest ( i , 1 ) if oddLen > maxLen : maxLen = oddLen bestStart = i - maxLen // 2 # check for even-length palindrome centered # between i and i+1 evenLen = mob . getLongest ( i , 0 ) if evenLen > maxLen : maxLen = evenLen bestStart = i - maxLen // 2 + 1 # extract the longest palindromic substring return s [ bestStart : bestStart + maxLen ] C# class GfG { public string longestPalindrome ( string s ) { Manacher mob = new Manacher ( s ); int n = s . Length ; // maximum length found so far int maxLen = 1 ; // starting index of longest palindrome int bestStart = 0 ; for ( int i = 0 ; i < n ; i ++ ) { // check for odd-length palindrome centered at i int oddLen = mob . getLongest ( i , 1 ); if ( oddLen > maxLen ) { maxLen = oddLen ; bestStart = i - maxLen / 2 ; } // check for even-length palindrome centered // between i and i+1 int evenLen = mob . getLongest ( i , 0 ); if ( evenLen > maxLen ) { maxLen = evenLen ; bestStart = i - maxLen / 2 + 1 ; } } // extract the longest palindromic substring return s . Substring ( bestStart , maxLen ); } } JavaScript class GfG { longestPalindrome ( s ) { const mob = new Manacher ( s ); const n = s . length ; // maximum length found so far let maxLen = 1 ; // starting index of longest palindrome let bestStart = 0 ; for ( let i = 0 ; i < n ; i ++ ) { // check for odd-length palindrome centered at i let oddLen = mob . getLongest ( i , 1 ); if ( oddLen > maxLen ) { maxLen = oddLen ; bestStart = i - Math . floor ( maxLen / 2 ); } // check for even-length palindrome centered // between i and i+1 let evenLen = mob . getLongest ( i , 0 ); if ( evenLen > maxLen ) { maxLen = evenLen ; bestStart = i - Math . floor ( maxLen / 2 ) + 1 ; } } // extract the longest palindromic substring return s . substring ( bestStart , bestStart + maxLen ); } } Code to Check If Substrings Are Palindromes C++ class GfG { public : // returns true/false for each query [l, r] vector < bool > checkPalindromes ( string s , vector < vector < int >>& queries ) { // preprocess the string using Mancher Class Manacher mob ( s ); vector < bool > res ; for ( auto & q : queries ) { int l = q [ 0 ], r = q [ 1 ]; // check if a[l..r] is a palindrome // using O(1) query res . push_back ( mob . check ( l , r )); } return res ; } }; Java class GfG { // returns true/false for each query [l, r] public ArrayList < Boolean > checkPalindromes ( String s , int [][] queries ) { // preprocess the string using Manacher Class Manacher mob = new Manacher ( s ); ArrayList < Boolean > res = new ArrayList <> (); for ( int [] q : queries ) { int l = q [ 0 ] , r = q [ 1 ] ; // check if a[l..r] is a palindrome // using O(1) query res . add ( mob . check ( l , r )); } return res ; } } Python class GfG : # returns true/false for each query [l, r] def checkPalindromes ( self , s , queries ): # preprocess the string using Manacher class mob = Manacher ( s ) res = [] for q in queries : l , r = q # check if a[l..r] is a palindrome # using O(1) query res . append ( mob . check ( l , r )) return res C# class GfG { // returns true/false for each query [l, r] public List < bool > checkPalindromes ( string s , int [,] queries ) { // preprocess the string using Manacher class Manacher mob = new Manacher ( s ); List < bool > res = new List < bool > (); // number of queries int qCount = queries . GetLength ( 0 ); for ( int i = 0 ; i < qCount ; i ++ ) { int l = queries [ i , 0 ]; int r = queries [ i , 1 ]; // check if a[l..r] is a palindrome // using O(1) query res . Add ( mob . check ( l , r )); } return res ; } } JavaScript class GfG { // returns true/false for each query [l, r] checkPalindromes ( s , queries ) { // preprocess the string using Manacher class const mob = new Manacher ( s ); const res = []; for ( const [ l , r ] of queries ) { // check if a[l..r] is a palindrome // using O(1) query res . push ( mob . check ( l , r )); } return res ; } } Comment Article Tags: Article Tags: Strings Pattern Searching DSA Arrays Microsoft palindrome + 2 More