Longest Palindromic Substring - Final Project

CS584 Algorithm Design And Analysis

- Deepa Varghese

Contents

1 Introduction

The longest palindromic substring or longest symmetric factor problem is the problem of finding a maximum-length contiguous substring of a given string that is also a palindrome, with applications in the field of physics and chemistry such as molecular symmetry. Here is an example of the longest palindromic substring - the longest palindromic substring of ”bananas” is ”anana”. The longest palindromic substring is not guaranteed to be unique; for example, in the string ”abracadabra”, there is no palindromic substring with length greater than three, but there are two palindromic substrings with length three, namely, ”aca” and ”ada”. This problem belongs to a class of problems relating to palindromic strings such as to return all maximal palindromic substrings (that is, all substrings that are themselves palindromes and cannot be extended to larger palindromic substrings) rather than returning only one substring or returning the maximum length of a palindromic substring, finding the longest palindromic subsequence etc.

2 Abstract

In this paper, there are three methods of finding the longest palindromic substring given any string, that are discussed, namely 1) Naive Brute Force technique 2) Dynamic Programming and 3) Manacher’s Algorithm. For each technique, the paper presents an explanation of the technique, followed by it’s psuedocode. We analyse the psuedocode independent of implementation and input, to understand its complexity. Following the theoretical analysis, we perform an empirical analysis of the algorithms, comparing and contrasting the results of the three algorithms.

3 Brute Force

The naive method of finding the longest palindromic substring of a given string s, is to find all possible substrings of the string s and checking if it is the longest one.

3.1 Algorithm Description

There are three loops being run, the outer-most loop, that is loop 1, fixes the left corner of the substring, loop 2 fixes the right corner of the substring. Thus the outer loops help in identifying the indexes of the left and right edges of the substring, consequently, identifying the actual substring. The innermost loop, loop 3 checks if the substring identified above is a palindrome. There are two ways to identify if a substring is a palindrome, by either looping through the first half of the string and checking if for each element it’s mirror image is equal to it (the method applied), or reversing the substring, storing it in a temporary location, and checking if the substring and it’s reverse are equal. By either way, if the substring is found to be a palindrome and it’s length is greater than the previously identified longest palindromic substring (this is set to an empty string at the beginning of the code), then it resets the result as being equal to the current substring being analysed. When all three loops finish, we get the longest palindromic substring in the string.

3.2 Psuedo-code

```
1) def BruteForce(Input):
2)    LongestPalindromicSubstring = ”” ... Initialise result to empty string
3)    For left = 0 to length(Input):
4)      For right = (left + 1) to length(Input):
```

```
5)          substring = Input[left...right]
6)          IsPalindrome = True
7)          For i = 0 to length(substring) / 2)):
8)             If substring[i] != substring[length(substr) - i - 1]:
9)                IsPalindrome = False
10)         If IsPalindrome and length(substring) > length(LongestPalindromicSubstring):
11)             LongestPalindromicSubstring = substring
12)    Return LongestPalindromicSubstring
```

3.3 Complexity

Time Complexity - In the above algorithm, for an input size of n the outer loop runs n times. The inner loop also runs for the entire length of the string, which is n. The innermost loop runs from 0 to half the length of the substring being analysed. Therefore, in the worst case,

$T(n) \leq \sum_{i=0}^{n} \sum_{j=i}^{n} \sum_{k=1}^{n/2} 1$
$T(n) \leq O(n^3)$
Therefore, $T(n) \in O(n^3)$

Space Complexity - For the Brute Force implementation, the auxillary complexity is $O(1)$

4 Dynamic Programming

To improve over the brute force solution, we first observe how we can avoid unnecessary re-computation while validating palindromes. Consider the case "ababa". If we already knew that "bab" is a palindrome, it is obvious that "ababa" must be a palindrome since the two left and right end letters are the same. The time complexity can be reduced by storing results of subproblems. The problem showcases the property of Optimal Substructure. For a substring A, assume that it has been proven to be a palindrome, we can find a substring longer than A by checking if the element before and after the said substring is equal. Thus forming a new substring B of $length(A) + 2$. B can further contribute in realising if substring C of length $length(B) + 2$ is a palindrome. Thus the longest Palindromic substring is obtained by combining the optimal results of shorter substrings.It also showcases the property of overlapping substucture, since it revisits the same sub-problem multiple times.

4.1 Algorithm Description

We maintain a Boolean table $DP[n][n]$ where n is the total length of the string. This table is filled in bottom up manner. For a substring in the Input string $Input[0...n]$, let i and j be the corner indices of the substring being analysed currently. For this substring we determine the value of the element $DP[i][j]$ in table DP. If the substring is a palindrome, it is set to 'True' otherwise 'False'. This verification i.e. checking that the substring is a palindrome, is done in the following manner. We first check the value of $DP[i+1][j-1]$, if the value is 'True' and $Input[i]$ is same as $Input[j]$, then we make $DP[i][j]$ is set to 'True'. Otherwise, the value of $DP[i][j]$ is made 'False'. Essentially, when $i == j$ that is a substring with a singleton character, we set the value as 'True' by default.

4.2 Psuedo-code

```
1)def DynamicProgramming(Input)
2)   DP[n][n] ... Create a 2D array of size $n^2$
3)   For i = 0 to length(Input)
4)     For j = 0 to length(Input)
5)       dp[i][j] = 'False' ... Initialise DP to 'False' for all elements
6)    lpsStartIndex = 0
7)    lpsEndIndex = 0
8)    For i = 0 to length(Input)
9)       start = i
10)      end = i
11)      while start $\geq$ 0:
12)         if start == end:
13)            dp[start][end] = True
14)         elif start + 1 == end:
15)            dp[start][end] = Input[start] == Input[end] ... substring of size = multiple of 2
16)         else:
17)            dp[start][end] = dp[start + 1][end - 1] and (Input[start] == Input[end])
18)
19)         if dp[start][end] and (end - start + 1) > (lpsEndIndex - lpsStartIndex + 1)
20)            lpsStartIndex = start
21)            lpsEndIndex = end
22)         start = start - 1
23)   return Input[lpsStartIndex ... lpsEndIndex + 1]
```

4.3 Complexity

Time Complexity - Initially we create a data structure such as a two dimensional array, of size $n * n$ where n is the size of the string and initialise the value of each element in the array in $O(n^2)$ time. This is followed by a For loop that runs n times, inside which we run a loop for half the size of the 2D array (only the bottom right part of the array gets filled). This results in a total time complexity of $O(n^2)$
Space Complexity - For dynamic programming, since we use create a data structure such as a two dimensional array of size $n * n$, the total space Complexity is $O(n^2)$. This space complexity acts as the major deterrent for the dynamic programming solution.

5 Manacher's Algorithm

Manacher (1975) found a linear time algorithm for listing all the palindromes that appear at the start of a given string. However, as it has been observed, the same algorithm can also be used to find all maximal palindromic substrings anywhere within the input string, again in linear time. Therefore, it provides a linear time solution to the longest palindromic substring problem. The algorithm inherently makes use of the symmetry property of a palindrome of being a mirror image around it's center, the expanding around it to find a palindromic substring.

To find in linear time a longest palindrome in a string, an algorithm may take advantage of the following characteristics or observations about a palindrome and a sub-palindrome:

- The left side of a palindrome is a mirror image of its right side.
- (Case 1) A third palindrome whose center is within the right side of a first palindrome will have exactly the same length as a second palindrome anchored at the mirror center on the left side, if the second palindrome is within the bounds of the first palindrome by at least one character (not meeting the left bound of the first palindrome). Such as "dacabacad", the whole string is the first palindrome, "aca" in the left side as second palindrome, "aca" in the right side as third palindrome. In this case, the second and third palindrome have exactly the same length.
- (Case 2) If the second palindrome meets or extends beyond the left bound of the first palindrome, then the distance from the center of the second palindrome to the left bound of the first palindrome is exactly equal to the distance from the center of the third palindrome to the right bound of the first palindrome.
- To find the length of the third palindrome under Case 2, the next character after the right outermost character of the first palindrome would then be compared with its mirror character about the center of the third palindrome, until there is no match or no more characters to compare.
- (Case 3) Neither the first nor second palindrome provides information to help determine the palindromic length of a fourth palindrome whose center is outside the right side of the first palindrome.
- It is therefore desirable to have a palindrome as a reference (i.e., the role of the first palindrome) that possesses characters farthest to the right in a string when determining from left to right the palindromic length of a substring in the string (and consequently, the third palindrome in Case 2 and the fourth palindrome in Case 3 could replace the first palindrome to become the new reference).
- Regarding the time complexity of palindromic length determination for each character in a string: there is no character comparison for Case 1, while for Cases 2 and 3 only the characters in the string beyond the right outermost character of the reference palindrome are candidates for comparison (and consequently Case 3 always results in a new reference palindrome while Case 2 does so only if the third palindrome is actually longer than its guaranteed minimum length).
- For even-length palindromes, the center is at the boundary of the two characters in the middle.

5.1 Psuedo-code

```
1)def ManachersAlgorithm(S)
2)  generate S' by inserting a bogus character ('#') between each character in S (including outer boundaries)
3)  Create array P to store the lengths of the palindrome for each center point in S (initially all 0s) (P.length = S'.length = 2*S.length+1)
4)  Track the following pointers (referencing indices in P and S'):
5)  R = the next element to be examined (initially 0)
6)  C = the largest/left-most palindrome whose right boundary is R-1 (initially 0)
```

```
7)  i = the next palindrome to be calculated (initially 1)
8)  L = character candidate for comparing with R. Computed implicitly as:L = i - (R - i)
9)  i' = the palindrome mirroring i from C. Computed implicitly as: i' = C - (i - C)
10)  While R < P.length:
11)     If i is within the palindrome at C (Cases 1 and 2):
12)        Set P[i] = P[i'] (else P[i] is set to 0)
13)        Expand the palindrome at i (primarily Cases 2 and 3; can be skipped in Case 1, though we have already shown that S'[R] != S'[L] because otherwise the palindrome at i' would have extended at least to the left edge of the palindrome at C):
14)        while S'[R] == S'[L]:
15)           increment P[i]
16)           increment R
17)     If the palindrome at i extends past the palindrome at C:
18)        Update C = i
19)     increment i
20)  return max(P)
```

5.2 Complexity

Time Complexity - As the algorithm is traversing through the newly created string which has the character'#' in between in each character and the front and end of the original string S (which has a length 2n+1, if the length of S is n), sequentially checking the center of the palindrome, it traverses through the entire list only once, to determine the longest palindromic substring. Therefore, the time complexity of the above algorithm is $O(n)$

Space Complexity - For any given string of length n, the algorithm creates a new string of length 2n+1. Therefore the space complexity is $O(n)$

6 Comparison

For the empirical analysis of each algorithm, and consequently comparing their performance amongst themselves, we vary the inputs provided to each algorithm, as explained in the following sections.

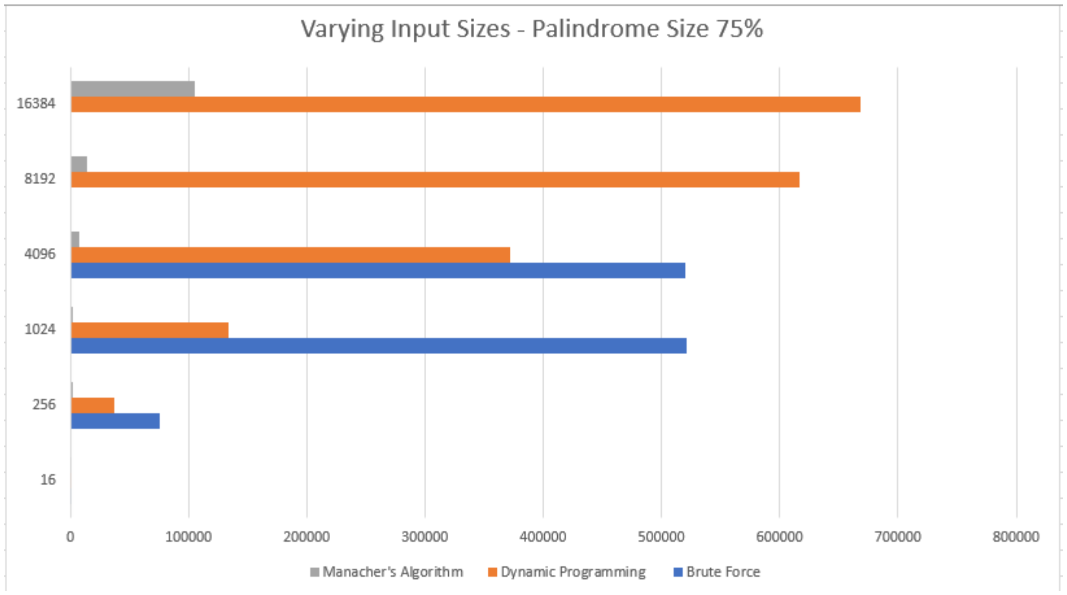
6.1 Varying Input Sizes

In this method, we compare the time taken by each algorithm for a different input sizes. The percentage of the input that should definitely be a palindrome can also be specified, but this value remains constant for all input sizes in one iteration of the experiment.

For this experiment, we defined a list of the following input sizes:

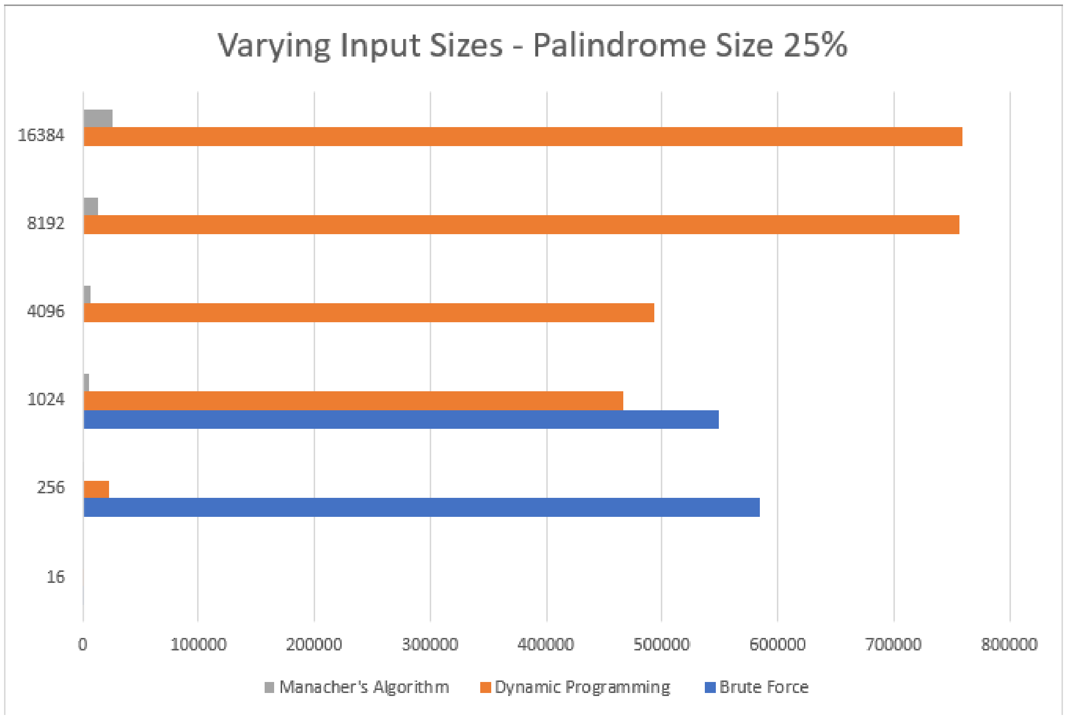
InputValues = [16, 256, 1024, 4096, 8192, 16384]

Case 1: There should be a palindrome inside the input of at least 75% of the length of the input:



NOTE:
X-Axis - Time taken to execute, in microseconds
Y-Axis - Length of Input String
Each input string had a palindrome of at least 75% of the total input size.

Case 2: There should be a palindrome inside the input of at least 25% of the length of the input:



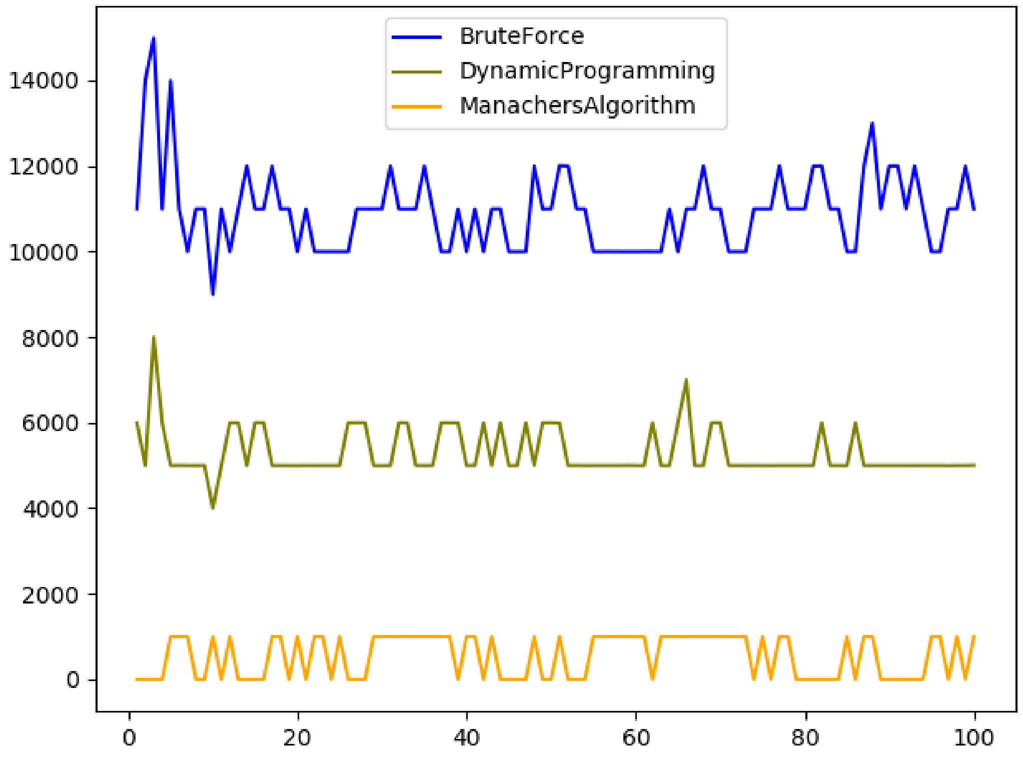
NOTE:

X-Axis - Time taken to execute, in microseconds
Y-Axis - Length of Input String
Each input string had a palindrome of at least 25% of the total input size.

6.2 Varying Palindromic sizes within same Input Size

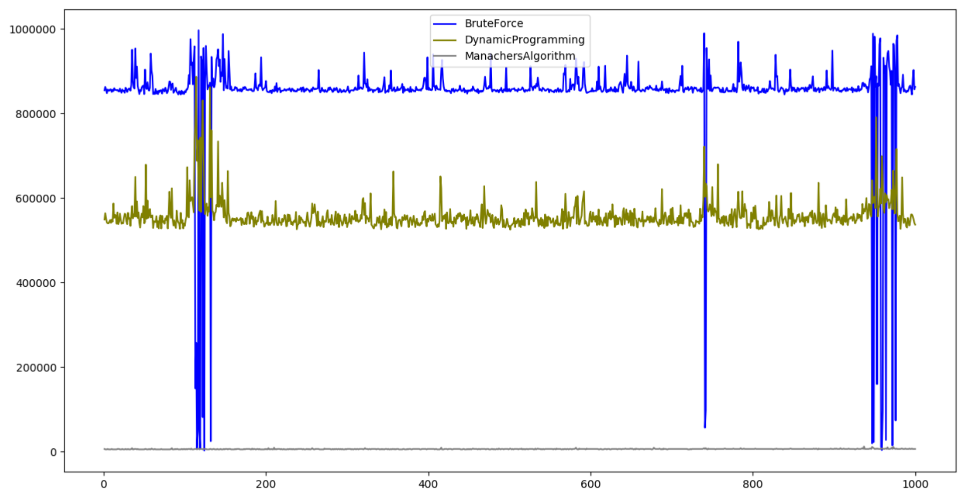
In this method, we compare the time taken by each algorithm for a fixed size input with a varied length of palindrome in it, gradually increasing the length of the palindrome as opposed to it's non-palindromic part of the input. The palindrome size starts from a small number and is incrementally increased by 1 step.
For very large input sizes, the system could not keep up with the computation required, or couldn't turn up with data that could be presented graphically for comparison. This might be due to the incremental increase by a small step from 1 to n that progressed excruciatingly slowly. Therefore, the results presented below are for small input sizes.

Case 1: For a fixed input size = 100, the palindrome size increases from 1 to 100 in each step.



NOTE:
X-Axis - Minimum expected length of the longest palindromic substring inside the given string
Y-Axis - Time take to execute, in microseconds

Case 2: For a fixed input size = 1000, the palindrome size increases from 1 to 1000 in each step.



NOTE:
X-Axis - Minimum expected length of the longest palindromic substring inside the given string
Y-Axis - Time take to execute, in microseconds

6.3 Observations:

1) There were a few instances where the Brute Force algorithm performed better than Dynamic. These must be the instances where the substring is judged as not being a palindrome in the very first step, which effectively brings down it's time complexity to n^2 for that instance, and without the overhead of the 2D array being used in Dynamic Programming, it tended to perform better.
2) Unsurprisingly, for very large inputs, the Brute Force algorithm always got interrupted with a timeout message.
3) For small palindromic sizes, the Brute Force algorithm and the Dynamic Programming algorithm performed worse as compared to strings with largers palindromic sizes.
4) The code was run in 2 different systems, with different processing power and memory available, and the results tended to vary according to them, where surprisingly Brute Force overtook Dynamic Programming. As observed by the comments posted by various users in Leetcode's discussion thread, the Dynamic Programming algorithm might suffer if the system cannot provide enough memory. Regardless of the system being used, Manacher's algorithm always outperformed the other two greatly.
5) There is a different Brute Force method that exists, that can perform in $O(n^2)$, where it traverses through all possible centers of palindrome in the given string. Think of growing a palindromic substring from the center by expanding outwards if the characters on both sides match. We can determine the length of a palindromic substring situated at a given center in $O(n)$ and there are only $2*(N-1)$ possible centers (one at each vertex and one between every two consecutive vertices). This will give us $O(n^2)$ time complexity with no additional space requirement. Thus being a better solution than Dynamic Programming.

7 Conclusion

Manacher's Algorithm undoubtedly performed better than the Dynamic Programming algorithm or the Brute Force, regardless of the type of input provided or the hardware resources. In order to choose Dynamic Programming algorithm we need to consider a time and space tradeoff, and theoretically the alternate Brute Force technique should perform better. Alternative linear time solutions were provided by Jeuring (1994), and by Gusfield (1997), who described a solution based on suffix trees. Efficient parallel algorithms also exist for this problem.

8 References

https://en.wikipedia.org/wiki/Longest-palindromic-substring
https://leetcode.com/articles/longest-palindromic-substring/
https://web.archive.org/web/20181208031518/https://articles.leetcode.com/longest-palindromic-substring-part-ii/
https://www.geeksforgeeks.org/longest-palindrome-substring-set-1/
https://kartikkukreja.wordpress.com/2016/10/16/problem-of-the-day-longest-palindromic-substring/