Finding Palindromic Substrings

Ermin Hodžić

Mar 2024

In this paper, we will explore algorithms for identifying palindromes within strings. While the problem of finding palindromes is currently not known to be particularly significant, it nevertheless represents an opportunity to explore searching with constraints that ultimately allow clever performance improvements. We also get the opportunity to showcase a wide variety of approaches to solving the problem. Methods that range from brute-force solutions, to clever but still slow algorithms, to using specialized and advanced data structures, to optimal custom-tailored algorithms – this problem has it all.

Requirements. No special requirements.

Contents

1 Palindromes

A **palindrome** is defined as a sequence of symbols that reads the same forwards as well as backwards. Examples of English words that are palindromes include "radar" and "level". There are also examples of full sentences that are palindromes if one ignores spaces and punctuation, such as "Was it a car or a cat I saw?" To make matters simple for us, in the rest of this tutorial, we will not make any such exceptions when considering whether a given string is a palindrome or not.

Determining whether a given string is a palindrome can be done in linear time, in a rather straightforward manner. From the definition of a palindrome, i.e. that it reads the same forwards as well as backwards, it is clear that determining whether a string S is a palindrome can be reduced to string matching. From the definition, S is a palindrome iff strings S and reverse(S) match without any errors. Algorithm 1 implements this without copying and reversing S, by keeping track of two indices i and i_{rev} that start from the opposite ends of the string and move towards the other side. If at any point the symbols at their positions disagree, then the string is not a palindrome. If the loop finishes without reporting that the string is not a palindrome, then the string must be a palindrome since every pair of corresponding mirrored positions has been validated.

Algorithm 1: isPalindrome(S)

Input: A string S
Output: **true** if S is a palindrome, and **false** if it is not

```
1 n = length(S)
2 for (i = 0; i ≤ n − 1) do
3     i_rev = n − 1 − i
4     if (S_i ≠ S_{i_rev}) then
5         return false
6 return true
```

Let us observe the execution of the algorithm on the palindrome "radar", with the compared symbols at every step indicated by red (starting from the left side) and blue (starting from the right side) colors, and in **bold**:

rada**r**
r**a**d**a**r
ra**d**ar
r**a**d**a**r
rada**r**
$\Longrightarrow$ return **true**

And similarly with the much longer string "wasitacaroracatisaw" (was it a car or a cat I saw):

wasitacaroracatisa**w**
w**a**sitacaroracatis**a**w
was**s**itacaroracati**s**aw
wasitacaroracatisaw
wasitacaroracatisaw
wasitacaroracatisaw
wasitacaroracatisaw
wasitacaroracatisaw
wasitacaroracatisaw
wasitacaroracatisaw
wasitacaroracatisaw
wasitacaroracatisaw
wasitacaroracatisaw
wasitacaroracatisaw
wasitacaroracatisaw
wasitacaroracatisaw
wasitacaroracatisaw

wasitacaroracatisaw
$\implies$ return **true**

Ok, from these two examples we can notice an obvious pattern of symmetry – once the two indices meet in the middle, they swap sides and repeat the exact same set of comparisons that they already performed in the first half, but with roles reversed. Can we prove this symmetry? Let us try.

From the definition, in order for a string to be a palindrome, it needs to read the same from both sides. However, if it reads the same from both sides, then it also reads the same from both sides when we restrict ourselves to only the first d characters (for some $d \leq |S|$). If this is true for $d = \lfloor \frac{n}{2} \rfloor$, then we know that the first half of the string reads the same as the reversed second half of the string. At this point, there is no need to keep going past the middle and compare the rest of the pairs of indices, because we already know that the two halves are each other's mirror images, and the rest of the pairs of symbols must match. The property that we just discovered is an important one that we will refer to later.

> ***Palindromic symmetry:*** Given a palindrome S, its left and right halves represent mirror images of each other, with possibly a single center character separating them if $|S|$ is an odd number.

Thus, we can rewrite our algorithm to be twice as fast by not going past the middle point.

Algorithm 2: isPalindrome(S)

Input: A string S
Output: **true** if S is a palindrome, and **false** if it is not

```
1 i = 0
2 i_rev = length(S) - 1
3 while (i < i_rev) do
4     if (S_i ≠ S_i_rev) then
5         return false
6     else
7         i++
8         i_rev --
9 return true
```

The execution of Algorithm 2 on the string "wasitacaroracatisaw" now looks like:

wasitacaroracatisaw
wasitacaroracatisaw
wasitacaroracatisaw
wasitacaroracatisaw
wasitacaroracatisaw
wasitacaroracatisaw
wasitacaroracatisaw
wasitacaroracatisaw
wasitacaroracatisaw
wasitacaroracatisaw
$\implies$ return **true**

We will return to the property of palindromic symmetry later when exploring efficient algorithms for detection of palindromic substrings.

2 Finding all Palindromic Substrings

In this section, we are going to explore the general problem of finding all substrings of a given string which are palindromes. For example, the string "bananas" has the following palindromic substrings (including duplicates):

- **Length 1:** "b", "a", "n", "a", "n", "a", "s"
- **Length 3:** "ana", "nan"
- **Length 5:** "anana"

Whereas the string "gimmicks" has the following ones:

- **Length 1:** "g", "i", "m", "m", "i", "c", "k", "s"
- **Length 2:** "mm"
- **Length 4:** "immi"

The number of all palindromic substrings can actually be quadratic with the length of the given string. Such an example is demonstrated by the string "aaaaaa", for which all palindromic substrings are:

- **Length 1:** "a", "a", "a", "a", "a", "a"
- **Length 2:** "aa", "aa", "aa", "aa", "aa"
- **Length 3:** "aaa", "aaa", "aaa", "aaa"
- **Length 4:** "aaaa", "aaaa", "aaaa"
- **Length 5:** "aaaaa", "aaaaa"
- **Length 6:** "aaaaaa"

2.1 Brute-Force Algorithm

Thinking in terms of the length of a palindromic substring, it is straightforward to come up with a brute-force algorithm that iteratively considers all possible lengths of palindromic substrings. Given a fixed length, such an algorithm could try all possible starting positions and report the palindromes that it finds in such a manner.

Algorithm 3: allPalindromicSubstrings(S)

Input: A string S
Output: A collection of all palindromic substrings of S

```
1 R = ∅
2 n = length(S)
3 for (l = 1; l ≤ n) do
4     for (i = 0; i ≤ n − l) do
5         if (isPalindrome(S[i...(i + l − 1)])) then
6             R.add(S[i...(i + l − 1)])
7 return R
```

The running time of this algorithm is $O(|S|^3)$, since it is checking every position and every length, and every such check takes time linear in the given length. In fact, the running time is $O(|S|^3)$ only if the collection R simply holds read-only information about the substrings (such as pairs of numbers representing the index of the original string at which the substring begins, and its length), without copying those substrings and allocating memory for them. If we needed to actually extract all the palindromic substrings into their own places in memory, that would increase the running time to $O(|S|^4)$, and would require up to $O(|S|^3)$ memory to store all of the possible $O(|S|^2)$ palindromic substrings. However, the last point is not specific to this algorithm, and holds true for all algorithms that aim to extract all palindromic substrings as separate strings in memory.

2.2 Quadratic Algorithm

Can we do better than the brute-force algorithm? We actually can, and one such way is to make use of the property of palindromic symmetry that we discovered in the previous section. To remind ourselves, the specific property stated that a palindrome consists of two mirrored halves and possibly a single center character if its length is an odd number. Thus, instead of iterating over all combinations of possible lengths and starting positions, and then performing worst-case linear-time palindrome checks; we could instead iterate over all possible center positions, and then expand the two mirrored halves as far as possible, as long as they match. The centre could be either a space between two characters (resulting in an even-length palindrome), or it could be a single character (resulting in an odd-length palindrome).

Let us see how this algorithm would perform on the string "ACBCAACBCAD". We are going to underline the current center considered, and color the current extent of the discovered palindromic substring by red.

$$
\begin{array}{c}
\underline{\text{A}}\text{CBCAACBCAD} \\
\text{A}\underline{\ }\text{CBCAACBCAD} \\
\text{A}\underline{\text{C}}\text{BCAACBCAD} \\
\text{AC}\underline{\ }\text{BCAACBCAD} \\
\text{AC}\underline{\text{B}}\text{CAACBCAD} \Longrightarrow \text{AC}\underline{\text{BC}}\text{AACBCAD} \Longrightarrow \text{AC}\underline{\text{B}}\text{CAACBCAD} \\
\text{ACB}\underline{\ }\text{CAACBCAD} \\
\text{ACB}\underline{\text{C}}\text{AACBCAD} \\
\text{ACBC}\underline{\ }\text{AACBCAD} \\
\text{ACBC}\underline{\text{A}}\text{ACBCAD} \\
\text{ACBCA}\underline{\ }\text{ACBCAD} \Longrightarrow \text{ACBCA}\underline{\ }\text{ACBCAD} \Longrightarrow \text{ACBCA}\underline{\ }\text{ACBCAD} \\
\Longrightarrow \text{ACBCA}\underline{\ }\text{ACBCAD} \Longrightarrow \text{ACBCA}\underline{\ }\text{ACBCAD} \Longrightarrow \text{ACBCA}\underline{\ }\text{ACBCAD} \\
\text{ACBCA}\underline{\text{A}}\text{CBCAD} \\
\text{ACBCAA}\underline{\ }\text{CBCAD} \\
\text{ACBCAA}\underline{\text{C}}\text{BCAD} \\
\text{ACBCAAC}\underline{\ }\text{BCAD} \\
\text{ACBCAAC}\underline{\text{B}}\text{CAD} \Longrightarrow \text{ACBCAA}\underline{\text{CBC}}\text{AD} \Longrightarrow \text{ACBCA}\underline{\text{ACBCA}}\text{D} \\
\text{ACBCAACB}\underline{\ }\text{CAD} \\
\text{ACBCAACB}\underline{\text{C}}\text{AD} \\
\text{ACBCAACBC}\underline{\ }\text{AD} \\
\text{ACBCAACBC}\underline{\text{A}}\text{D} \\
\text{ACBCAACBCA}\underline{\ }\text{D} \\
\text{ACBCAACBCA}\underline{\text{D}}
\end{array}
$$

We see that this algorithm will indeed find all palindromic substrings, including "nested" palindromes that share the same center, because it starts from the shortest and expands as far as it can. As soon as a mismatch is found, there would be no point in trying to expand further from the current center, as the two sides would no longer be mirror images of each other. Algorithm 4 shows the implementation of this procedure.

The variable c iterates over all possible centers, with its even values representing characters of S, and its odd values representing spaces between two consecutive characters of S. If c is an even number, then it is centered on a character, and the bounds (within S) of a palindrome centered on c whose two halves each have length d would be $i - d$ and $i + d$. If c is an odd number, then the bounds would be $i - d + 1$ and $i + d$. In order to avoid branching with an if statement, we make use of the binary variable $isSpace$, to adjust the bounds correctly based on the parity of c. There have been confirmations of the existence of such humans who are arguing for reducing every "centered on space" case to a "centered on a character" case, by inserting some character that is not found in the string, between every pair of consecutive characters of the original string, and then simply treating every position as centered on a character. However, I do not like the idea of editing the original string or allocating additional memory to store a temporary string of twice the length, when very simple index adjustments via a binary parity variable ($isSpace$) will do the job just fine in a single pass.

The running time of this algorithm is $O(|S|^2)$ to report all $O(|S|^2)$ possible palindromic substrings, which is a significant improvement over the previous brute-force method. For the problem of finding all palindromic substrings, this represents the most efficient method in the worst case, as there could be exactly $O(|S|^2)$ palindromic substrings, and enumerating and reporting them all cannot take less time than that (simply counting their total number, however, can). Thus, when exploring algorithms for finding palindromic substrings that are more efficient than the quadratic algorithm in the worst case, we have to restrict the problem to something with a smaller output size, such as finding

Algorithm 4: allPalindromicSubstrings(S)

Input: A string S
Output: A collection of all palindromic substrings of S

```
1  R = ∅
2  n = 2 * length(S) − 1 ;                    // |S| characters and |S| − 1 spaces between them
3  for (c = 0; c < n) do
4      i = ⌊c/2⌋ ;                            // Obtain the character index within the string
5      isSpace = c − 2 * i ;  // 1 if c is odd and the center is a space, and 0 if it is even
         and the center is a character
6      if (isSpace = 0) then
7          R.add(S[i...i])
8      d = 0 ;                                // The current length of the palindrome's sides
9      while (i + d + 1 < n   and   i − d − 1 + isSpace ≥ 0   and   S_{i+d+1} = S_{i−d−1+isSpace} ) do
10         d++
11         R.add(S[(i − d + isSpace)...(i + d)])
12 return R
```

the longest palindromic substring(s).

3 Finding The Longest Palindromic Substring

In this section, we are going to explore the problem of finding the single longest palindromic substring. Any of the previously presented algorithms can be used to find all palindromic substrings, and then we can simply select the longest of them. However, the worst case running time of such an algorithm would be $\Omega(|S|^2)$. If we give up on enumerating all palindromic substrings and focus only on finding the longest, we can accomplish that task much faster. In the following subsections, three different algorithms will be presented. Two of them will be based on data structures that are applicable to general string searching, adapted to our specific problem, and the last one will be a linear-time algorithm that is specific to the problem of finding the longest palindromic substrings around each possible center.

3.1 Hash-Based Binary Search Algorithm

A hash table is a data structure that allows one to store objects inside it, and answer queries of the kind "Does this object already exist inside the hash table?" in $O(1)$ average running time if we give it an already-computed hash value of the object. If the previous sentence has left you confused, please read up on hash tables before returning to this subsection, as going into detail about the inner workings of hash tables is beyond the scope of this article. Moving forward, I will assume that the reader is familiar with hash tables. Now, let us leave hash tables aside for a moment and go back to our problem of finding the longest palindromic substring.

Assume that we have a function that allows us to answer the following **yes/no** question: *Given an integer* $d \in [0, \lfloor \frac{|S|}{2} \rfloor]$*, does the string S contain a palindromic substring whose mirrored sides are of length that is at least d?* If we did have such a function, then we could solve the problem of finding the longest palindromic substring by finding the largest integer $d \in [0, \lfloor \frac{|S|}{2} \rfloor]$ for which the function would return a positive answer. In an example of the string "gimmicks", such a function would return **no** for $d \in \{3, 4\}$, and **yes** for $d \in \{0, 1, 2\}$ – any single letter substring would be an example of a side-length-0 palindrome, "mm" would be the single side-length-1 palindrome, and "immi" would be the single side-length-2 palindrome. Because of palindromic symmetry, such a function would return **yes** for every non-negative integer that is smaller than half the length (floor) of the longest palindromic substring. This is because the existence of a palindromic substring whose side-length is d implies existence of nested palindromic substrings around the same center of every side length between d and 0. Thus, if the longest palindromic substring is of length l, our function is going to report **yes** for all integers between 0 and $\lfloor \frac{l}{2} \rfloor$, and **no** for all integers that are greater. This means that because of the nested palindromic symmetry, the function has the property that it is monotone and "continuous" on the interval of integers between 0 and $|S|$ (as its input value increases, it returns a sequence of **yes** values followed by a sequence of **no** values, with no interruptions anywhere and without mixing the return values between the yes/no camps), and that allows us to use binary search. Algorithm 5 shows the idea.

Algorithm 5: longestPalindromicSubstring(S)

Input: A string S
Output: The longest palindromic substring of S

```
1 lb = 0
2 ub = ⌊length(S)/2⌋
3 while (lb ≤ ub) do
4     d = (lb + ub)/2
5     if ( hasPalindromeOfSideLength(S, d) ) then
6         lb = m + 1
7     else
8         ub = m − 1
  /* ub holds the longest side length at the end of the while loop          */
9 return extractPalindromeOfSideLength(S, ub)
```

The while loop takes $O(\log |S|)$ steps to find the upper bound, so the total running time of this algorithm depends on how efficient functions $hasPalindromeOfSideLength(S, d)$ and $extractPalindromeOfSideLength(S, ub)$ are, and will be an $O(\log |S|)$ multiple of their running time. So how quickly can those functions be implemented?

If we had a hash table of every d-length substring of S, for each length d that we are querying, then the function $hasPalindromeOfSideLength(S, d)$ could run in linear time. For every center position (every space and every

letter), we would check whether the substring formed by reverse-reading the d characters to the left of the center is found in the hash table at the same center position. If yes, then we have identified two mirror sides of length d each, and we can return **yes**. Next, we will see how to efficiently compute the hash of each such substring of a particular length in total $O(|S|)$ time for all substrings, that would then be checked against the hash table.

3.1.1 Extracting Hash Values of Each Substring of a Given Length

Our scenario is as follows. We are given a string S, and an integer k. Our task is to compute the hash values of each length-k substring of S (also called a k-mer), so that the substrings can be inserted into a hash table.

First, we need to decide on a hashing function that would encode our strings as numbers, and there are many possibilities. Going into how to construct a good string hashing function that minimizes collisions is beyond the scope of this tutorial, so we will not do that. What we will need to ensure, however, is that whatever hashing function is chosen, it allows us to efficiently compute the hash value of the k-mer at position i if we already know the hash value of the k-mer at position $i-1$. One such class of hash functions would be the polynomial hash, which treats the (numeric representation of) characters of a string as coefficients of a polynomial. More specifically, for a string A of length k, and two positive integers p and m, the polynomial hash of A is calculated as:

$$H_{p,m}(A) = (\sum_{i=0}^{k-1} ascii(A_i) \cdot p^i) \mod m$$

The following shows the polynomial hash values for all 4-mers of the string S = "gimmicks", and $p = 271, m = 108263$.

gimmicks $\rightarrow$ $H_{p,m}(\text{gimm}) = (s_0 271^3 + s_1 271^2 + s_2 271 + s_3) \mod 108263 = 53008$
gimmicks $\rightarrow$ $H_{p,m}(\text{immi}) = (s_1 271^3 + s_2 271^2 + s_3 271 + s_4) \mod 108263 = 94480$
gimmicks $\rightarrow$ $H_{p,m}(\text{mmic}) = (s_2 271^3 + s_3 271^2 + s_4 271 + s_5) \mod 108263 = 21866$
gimmicks $\rightarrow$ $H_{p,m}(\text{mick}) = (s_3 271^3 + s_4 271^2 + s_5 271 + s_6) \mod 108263 = 51273$
gimmicks $\rightarrow$ $H_{p,m}(\text{icks}) = (s_4 271^3 + s_5 271^2 + s_6 271 + s_7) \mod 108263 = 9116$

From the given example, you can probably notice how polynomial hash allows us to compute the hash value of the next k-mer in $O(1)$ time. For example, given $H_{p,m}(\text{gimm})$, if we subtract its first polynomial term, we will obtain $H_{p,m}(\text{imm}) = (s_1 271^2 + s_2 271 + s_3) \mod 108263$. If we now multiply $H_{p,m}(\text{imm})$ by p, and then add s_4 to it, and finally take the result modulo m, we will obtain $H_{p,m}(\text{immi}) = (s_1 271^3 + s_2 271^2 + s_3 271 + s_4) \mod 108263$. Thus, we can move from having the hash value of one k-mer to obtaining the hash value of the next by applying one modular subtraction, one multiplication and one addition, taking a total of $O(1)$ time. Thus, the total time to compute hash values of all k-mers of a string S is going to be $O(|S|)$. Algorithm 6 shows an $O(|S|)$ implementation of the above-described procedure.

Algorithm 6: hashKmers(S, k, p, m)

Input: A string S
Output: A hash table of the string's k-mers

```
1  HT = constructHashTable(m) ;          // The table will hold indices of inserted k-mers
2  h = 0 ;                                // The hash value of the current k-mer
3  pk = 1 ;                               // This variable will hold p^k
4  for (i = 0; i < k) do
5      h = (h · p + ascii(s_i)) mod m
6      pk = (pk · p) mod m
7  HT.insert( (h, 0) )
8  for (i = k; i < length(S)) do
9      h = (h · p + ascii(s_i)) mod m ;                    // Add the new character
10     h = (h − s_{i−k} · pk) mod m ;      // Remove the leftmost character that is outside the
       current sliding window
11     HT.insert( (h, i − k + 1) )
12 return HT
```

3.1.2 The Full Algorithm

So far, we have described a binary-search-based strategy, and we have seen how to efficiently construct a hash table for all substrings of some given length k in linear time. Algorithm 6 shows how to hash all forward-reading k-mers, which would represent the right side of a palindrome. What remains to show is how to query such a hash table against reverse-read k-mers, which would represent the left side of the palindrome.

To such purpose, we can apply nearly the exact same method that Algorithm 6 uses, only in this case we would be computing hashes in reverse – starting from the end of the string and going leftwards; the term that we would be subtracting (line 10) would be $s_{i+k} \cdot pk$ instead of $s_{i-k} \cdot pk$; and the index of the hashed k-mer (line 11) would be i instead of $i + k - 1$. Once a hash of a possible left side of a palindrome has been obtained, we would check whether the hash table has a k-mer stored with that exact same hash, and whether its index is either $i + 1$ (in the case of a center that is a space between letters), or $i + 2$ (in the case of a center that is a letter). If they match, then the function can return **yes**; and if the for loop has reached the end without reporting **yes**, then the function can return **no**. I will omit the pseudocode listing for this just-explained $hasPalindromeOfSideLength$ function due to minor differences relative to Algorithm 6.

The last thing to discuss would be what the function $extractPalindromeOfSideLength$ would look like, which we would need to run once we have obtained the side length of the longest palindromic substring. This function can be written as just a slight variation of $hasPalindromeOfSideLength$ which would be called with the final result of the binary search as its parameter k. Upon finding a successful match with a letter center (prioritizing these because they are longer by 1), it would simply extract and return that substring. If there is none, then it would return the first one that it finds with a space center. The following illustrates the comparison of the backward hashing sweep (red) vs the computed hash table of the forward k-mers (blue) for the two center cases:

Center as a space: AAAAAAAAAAAB

Center as a letter: AAAAAAAAAAAB

Above, the backward-read side's (in red) hash value is queried twice against the hash table for its possible mirror images (in blue), for the two possibilities of a palindromic center. In the above case, both queries return successful matches, and the final result would be the second one because it results in the longer palindrome. The final algorithm would then look like Algorithm 7.

Algorithm 7: longestPalindromicSubstringByHashing(S, p, m)

Input: A string S, and two positive integers p and m
Output: The longest palindromic substring of S

```
1  lb = 0
2  ub = ⌊length(S)/2⌋
3  while (lb ≤ ub) do
4      d = (lb + ub)/2
5      HT = hashKmers(S, d, p, m)
6      if ( hasPalindromeOfSideLength(S, d, HT, p, m) ) then
7          lb = m + 1
8      else
9          ub = m − 1
   /* ub holds the longest side length at the end of the while loop        */
10 HT = hashKmers(S, ub, p, m)
11 return extractPalindromeOfSideLength(S, ub, HT, p, m)
```

To analyze the running time of our final algorithm, we note that the binary search loop performs $O(\log |S|)$ steps to find the upper bound. Each such step involves creating a hash table through a linear forward sweep, and then performing a backward sweep during which the hash table is queried. The execution of the backward sweep stops the moment that a match is found. Once the loop has finished, one more hash table is created and one more backward sweep is performed in order to extract the longest palindromic substring. The total running time is thus $O(|S| \log |S|)$, and the largest amount of memory that we need to have allocated at any given moment is just $O(|S|)$ (under the assumption that the size of the hash table is $O(|S|)$).

3.2 The Suffix-Array-Based Algorithm

Algorithm described in this section will make use of the *Suffix Array* data structure to enable asymptotically optimal discovery of the longest palindromic substring. Explaining how the suffix array works, how it is constructed in linear time, and proving certain properties of it and its associated longest common prefix (LCP) array, warrants a whole paper of its own. Describing all of that is beyond the scope of the current paper, so I will continue with the text under the assumption that you are familiar with the notion and inner workings of a (generalized) suffix array. I will, however, provide the definitions of the tools that we will be using, to ensure that we are all on the same page before moving forward.

> ***Suffix array:*** For a given string S, a **suffix array** SA is an integer array of equal length to that of S, where the i-th element of the suffix array, SA_i, contains the index of the suffix of S that ranks as i-th in the sorted (lexicographic) order of all suffixes of S.

Let us digest the above definition by going through an example of what a suffix array of the string "bananas" would look like. First, let us write down all the suffixes of the string S = "bananas", where we denote the i-th suffix (the one that starts from the i-th character of S) by S_i. We will follow the common practice of appending a unique character to the end of the string, one that does not otherwise appear inside the string and which is lexicographically smaller than any character of S. We will use the \$ sign for it.

$$\begin{aligned} S_0 &= \text{bananas\$} \\ S_1 &= \text{ananas\$} \\ S_2 &= \text{nanas\$} \\ S_3 &= \text{anas\$} \\ S_4 &= \text{nas\$} \\ S_5 &= \text{as\$} \\ S_6 &= \text{s\$} \\ S_7 &= \text{\$} \end{aligned}$$

Let us now sort the suffixes in ascending lexicographic order.

$$\begin{aligned} 0:&\quad S_7 = \text{\$} \\ 1:&\quad S_1 = \text{ananas\$} \\ 2:&\quad S_3 = \text{anas\$} \\ 3:&\quad S_5 = \text{as\$} \\ 4:&\quad S_0 = \text{bananas\$} \\ 5:&\quad S_2 = \text{nanas\$} \\ 6:&\quad S_4 = \text{nas\$} \\ 7:&\quad S_6 = \text{s\$} \end{aligned}$$

The suffix array of the string S = "bananas\$" would then be the reading of the indices of the sorted suffixes, i.e. $SA(\text{"bananas\$"}) = (7, 1, 3, 5, 0, 2, 4, 6)$. Additionally, given a string and its suffix array, the longest common prefix array (LCP) of the same string can be constructed.

> ***Longest common prefix array:*** Given string S, the *longest common prefix array* LCP is an integer array of size $|S| - 1$, where the i-th element of the LCP array, LCP_i, contains the length of the longest string that is prefix of both of the two suffixes in sorted positions i and $i - 1$.

Refering to the list of sorted suffixes of the string "bananas\$" above, we can see that its LCP array would be:

$$\begin{aligned} LCP_1 &= LCP(\text{\$}, \text{ananas\$}) &&= 0 \\ LCP_2 &= LCP(\text{ananas\$}, \text{anas\$}) &&= 3 \\ LCP_3 &= LCP(\text{anas\$}, \text{as\$}) &&= 1 \\ LCP_4 &= LCP(\text{as\$}, \text{bananas\$}) &&= 0 \\ LCP_5 &= LCP(\text{bananas\$}, \text{nanas\$}) &&= 0 \\ LCP_6 &= LCP(\text{nanas\$}, \text{nas\$}) &&= 2 \\ LCP_7 &= LCP(\text{nas\$}, \text{s\$}) &&= 0 \end{aligned}$$

Combining the suffix array and the LCP in the following table, we obtain:

Rank	Suffix	SA	LCP
0	$	7	-
1	ananas$	1	0
2	anas$	3	3
3	as$	5	1
4	bananas$	0	0
5	nanas$	2	0
6	nas$	4	2
7	s$	6	0

Generally, having a suffix array and its corresponding LCP array of a string S allows us to efficiently perform a wide range of variations of (applications of) string searching within S. I.e., we can use it to perform searching for specific substrings of S with certain desired properties, whether that they match a given query string P, or perhaps something like the problem that we have at hand – looking for the longest palindromic substring.

Many of the applications of suffix arrays are enabled by constructing what is known as a *Generalized Suffix Array*, which allows us to compare suffixes of multiple strings at once, combined. That is achieved by simply building a suffix array of a string obtained by concatenating multiple strings together, but putting unique delimiters between them in order to differentiate between otherwise-identical suffixes but that originate in different strings. Let us look at an example of a suffix array of two strings "banana" and "banned", whose concatenation would be "banana$banned#":

Rank	Suffix	SA	LCP
0	#	13	-
1	$banned#	6	0
2	a$banned#	5	0
3	ana$banned#	3	1
4	anana$banned#	1	3
5	anned#	8	2
6	banana$banned#	0	0
7	banned#	7	3
8	d#	12	0
9	ed#	11	0
10	na$banned#	4	0
11	nana$banned#	2	2
12	ned#	10	1
13	nned#	9	1

Having a generalized suffix array allows us not only to efficiently perform string search queries within multiple strings, but it also allows us to efficiently compare and spot similarities of suffixes of multiple strings for whatever purpose we might need. For example, LCP_7 tells us that there is a length-3 substring "ban" that is shared by both of our input strings, since that is the LCP of the two suffixes that are ranked 6 and 7, which originate in different strings.

Moving on, a property that suffix array provides and that we will need for our application is the following claim that really applies to any general lexicographically sorted collection of strings, but we are working with suffix arrays here so we are discussing it in this context.

Claim 1. *Given a string S, its suffix array, and a suffix of rank i: (1) there is no lexicographically smaller suffix of S that has a longer common prefix with it than the one ranked $i-1$ (if any exists); and (2) there is no lexicographically larger suffix of S that has a longer common prefix with it than the one ranked $i+1$ (if any exists).*

This claim can actually be generalized for not just two suffixes of consecutive ranks, but to any pair of suffixes, as follows. (I will provide proofs of these claims once/if I write a paper about Suffix Arrays.)

Claim 2. *Given a string S, its suffix array, and two suffixes whose ranks are i and j ($i < j$): (1) there is no suffix of S that is ranked lower than i that has a longer common prefix with the one ranked j; and (2) there is no suffix of S that is ranked higher than j that has a longer common prefix with the one ranked i.*

In other words, if we denote the longest common prefix between two suffixes ranked i and j by $LCP(i, j)$, then the following holds:

$$\forall k < i : LCP(k, j) \leq LCP(i, j)$$
$$\forall l > j : LCP(i, l) \leq LCP(i, j)$$

Lastly, the following claim describes the exact length of the longest common prefix of any two arbitrary suffixes as the minimum of LCPs of all pairs of suffixes of consecutive ranks between them.

Claim 3. *Given a string S, its suffix array, and two suffixes whose ranks are i and j (i < j):*

$$LCP(i, j) = \min_{i<k\leq j} \{LCP_k\}$$

Now that we have described the needed tools, let us go back to our problem of finding the longest palindromic substring – how does having a suffix array help us? Well, let us first think about what we are exactly looking for in the context of suffixes. If we are examining a possible palindromic center that is a space between indices i and $i + 1$, then to determine to length of the longest palindromic substring centered on that space, we would want to know the longest common prefix of the suffix S_{i+1} and the equivalent suffix of the index i in $rev(S)$, which is $rev(S)_{|S|-1-i}$; and if the center is a letter at index i instead, then we are looking for LCP of S_{i+1} and $rev(S)_{|S|-i}$. As mapping indices between S and $rev(S)$, and a generalized suffix array constructed on their concatenation is straightforward, our problem is then reduced to efficiently computing the LCP of $O(|S|)$ pairs of indices (for each possible space and letter palindromic center) within the suffix array. Claim 3 tells us how to compute it: to get the LCP for suffixes i and j, we have to find the minimum LCP value in the range $[i, j]$ (assuming that $i < j$) of the LCP array. However, applying that formula naively could result in $O(|S|^2)$ running time.

Let us formally write the new, equivalent, problem that we now need to solve:

> ***Range Minimum Query (RMQ) problem:*** Given an array of numbers A of size n, and a series of queries of the form (i, j) where $0 \leq i \leq j \leq n - 1$, find and report the minimum value of A that is between indices i and j for each query.

A variant of this problem is the *static* RMQ problem, named so because the underlying array on which the queries are performed (which is our LCP array) does not ever change between queries. The great news for us is that in the case of the static RMQ problem, it is possible to build a data structure in $O(|S|)$ time, which supports queries in $O(1)$ time. Just like in the case of how to build a suffix array, I have to hit you with a disclaimer that this is a paper on finding palindromic substrings, and not an advanced data structure paper, so I will not provide details of how to construct such a data structure here (if you are curious, look up *sparse tables*). However, I do plan to *eventually* write a paper on range queries (both dynamic and static), so stay tuned.

The above allows us to complete the task of finding the longest palindromic substring in a total of $O(|S|)$ time, through the following steps:

1. Concatenate the original string S with $rev(S)$, with a unique delimiter between them.
2. Build a generalized suffix array of the two strings, and compute the LCP array and the sorted rank of each index of the two strings.
3. Construct a data structure that is built in $O(|S|)$ time and supports range minimum queries on the LCP array in $O(1)$ time each.
4. For each possible palindromic center (a letter, or a position between two consecutive letters), compute the starting indices of the right and left side in S and $rev(S)$, determine their sorted ranks, and report their LCP in $O(1)$ time using the precomputed data structure.
5. The largest reported value in the previous step will give away the center of the longest palindromic substring of S.

3.3 Direct Linear-Time Algorithm

At last, we will now look into the optimal algorithm for detection of the longest palindromic substring, which will not be using any advanced broad-purpose data structures (such as hash tables or suffix arrays), but rather tackle the problem directly. In a sense, this algorithm represents an extension of the quadratic algorithm that was given and discussed in Section 2.2; the extension coming in the form of reuse of already-processed information, ultimately allowing an improvement from a quadratic-time to a linear-time algorithm.

As a reminder, the quadratic time algorithm of Section 2.2 tries every possible center (every letter, and every position between two consecutive letters), and expands from it equally left and right for as long as the two characters at the two boundaries of the expansion match. As the result, the longest palindromic substring will be found for every tried center position. To begin to see how we can reuse some of that information to speed the algorithm up, let us examine a few examples of palindromic substrings. The following is one palindromic substring of some string, centered on the letter 'd', with its left side colored red and its right side colored blue.

...bdacbdqhjzdzjhqdbcafa..

Is there anything we can conclude from this palindromic substring that might speed up the remainder of our search? Well, one thing to notice is that aside from its own center which is the letter 'd', there is no other valid palindromic center contained within this palindrome with a non-zero-length side. Thus, if we can somehow know this information, we can skip trying all the candidate centers contained within it. However, that would not be much of a speedup, since the inner loop of the quadratic algorithm would halt immediately for all these candidate centers anyway. What about the case when there *are* valid non-zero-length side palindromic centers contained within? Let us examine an example of one such center contained within our palindrome, colored green.

...bdacbtktbqhjzdzjhqbtktbcafa..

The letter 'k' is the center of the palindrome "btktb". The interesting thing to note here is that since "btktb" is fully contained within the sides of our palindrome (as opposed to being nested inside it, on the same center 'k'), and because it iself is a palindrome, it will be read exactly the same on either side of our palindrome. I.e. both sides of our palindrome contain the string "btktb", read that exact same way from both left to right, as well as right to left. Moreover, since the palindrome "btktb" is completely contained inside our palindrome, and does not extend beyond it, that means that in our search for the longest palindromic substring, we can safely skip the palindromic centers 'k' that the two palindromes "btktb" are centered on – because they cannot possibly have longer length than the palindrome that they are contained within, if we are somehow provided the information about how far its palindromes extend and whether they are fully contained inside the one we just examined. In contrast, the quadratic algorithm would have tried both these centers and spent a few iterations of the inner loop working on them.

In the paragraph above, it was pointed out that the two centers can be safely skipped in our search for the longest palindromic substring because they do not extend beyond the borders of the current palindrome. What happens if they touch the border (note that it is not possible that they extend beyond, as then our current palindrome would not be maximal for the current center), such as in the following example?

...btktbacqhjzdzjhqcabtktb..

Now we can no longer discard the centers 'k', unless we know more about the rest of the string beyond our palindrome. We know that the two characters just outside the boundary of our greater palindrome do not match (otherwise, we would have extended to include those letters as well), but we can not say with certainty that the "k centers" are not valid centers of another, possibly even longer, palindrome. For example:

...gbtktbacqhjzdzjhqcabtktbac..

Here we see that extending around the left 'k' center would give us "gbtktba", which is not a palindrome. However, extending around the right 'k' center would give us "cabtktbac" which *is* a valid palindrome, and possibly extends even further. We thus arrive at the following rule:

Claim 4. *Given a palindromic substring that is maximally-extended around its center, any maximally-extended palindromic substring that is properly contained within it, and does not extend to or past its boundary, cannot be a center of a palindromic substring of an equal or greater length.*

Based on Claim 4, we only need to check palindromic centers that touch or extend beyond the boundary of our current palindrome; every other possible center can be safely skipped. Moreover, for each such properly-contained and maximally-extended palindrome within our current one that touches its border, we only need to continue its expansion past the border of our current palindrome – there is no need to check the inner characters again since we already know that they all match. In our previous example,

...gbtktbacqhjzdzjhqcabtktbac..

we would continue expanding the palindrome "btktb" past the letters 'b', thus skipping repetition of work we have already done.

The above-described properties and rules are all we need to devise our linear-time algorithm for finding the longest palindromic substring. What follows is a description of a variation of *Manacher's algorithm*.

3.3.1 Manacher's Algorithm

We are given a string S, in which we are to find the longest palindromic substring. We begin by constructing an array L of size $2|S| - 1$, in which we will fill in the length of the longest palindromic substring centered on each of the $2|S| - 1$ possible centers (each letter and each position between two consecutive letters). At the end, the longest palindromic substring will be fully described by the maximum value of the array L, giving the length of the palindrome as well as the position of its center.

The algorithm sweeps through all possible palindromic centers of S, from left to right. At each possible center indexed as i, the algorithm maximally extends left and right until the characters no longer match, and writes down the length of the detected palindromic substring as L_i. Given that all the $L_j (j < i)$ values are already computed, the algorithm then begins to copy all the $L_{i-1}, L_{i-2}, ...$ values that fall within the left side of the current palindrome, into mirror-copy entries $L_{i+1}, L_{i+2}, ...$ that fall within the right side of the current palindrome. The copying stops at the index of the first encountered L value which would indicate that the palindrome centered around it touches or extends the boundary of the current palindrome, and the whole procedure then repeats with that index as the center, with the extension continuing from the range that touches the current boundary. If no such index is found, the algorithm copies all the values and continues the search with the next center past the right-hand-side boundary of the current palindrome.

Let us illustrate the algorithm on the string example "gbtktbadabtktb". For simplicity sake, we will insert an underscore character between each letter, to denote the possible centers between consecutive letters, and make the lengths of S and L the same. Thus, our string S will be "g_b_t_k_t_b_a_d_a_b_t_k_t_b", and we begin with the following:

Idx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
S:	g	_	b	_	t	_	k	_	t	_	b	_	a	_	d	_	a	_	b	_	t	_	k	_	t	_	b
L:	1																										

The first letter, 'g', gives us the palindrome "g" of length 1. The side lengths are 0, so there is nothing to copy. This continues for the next few positions before the first letter 'k'.

Idx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
S:	g	_	b	_	t	_	k	_	t	_	b	_	a	_	d	_	a	_	b	_	t	_	k	_	t	_	b
L:	1	0	1	0	1	0	5																				

At center 'k' (index 6), we have the palindrome "btktb", which covers indices 2 to 10. Now, we start copying everything from values indexed 5 to 2, into values indexed 7 to 10. The first value whose range touches the border will be the one at index 10, centered on the letter 'b'.

Idx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
S:	g	_	b	_	t	_	k	_	t	_	b	_	a	_	d	_	a	_	b	_	t	_	k	_	t	_	b
L:	1	0	1	0	1	0	5	0	1	0	1+																

The length of the palindrome centered on 'b' is at least 1 (denoted as "1+"), so that is going to be the next valid center, and every position before it can be ignored as per Claim 4. The expansion of it fails immediately, and we continue with alternating ones and zeroes until we hit the letter 'd'.

Idx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
S:	g	_	b	_	t	_	k	_	t	_	b	_	a	_	d	_	a	_	b	_	t	_	k	_	t	_	b
L:	1	0	1	0	1	0	5	0	1	0	1	0	1	0	13												

This palindrome covers the range $(2, 26)$, so we start mirror-copying the values from the left into the right side, and we stop once we reach position 22, which represents the center of a palindrome that touches the border of the current one.

Idx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
S:	g	_	b	_	t	_	k	_	t	_	b	_	a	_	d	_	a	_	b	_	t	_	k	_	t	_	b
L:	1	0	1	0	1	0	5	0	1	0	1	0	1	0	13	0	1	0	1	0	1	0	5+				

Idx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
S:	g	_	b	_	t	_	k	_	t	_	b	_	a	_	d	_	a	_	b	_	t	_	k	_	t	_	b
L:	1	0	1	0	1	0	5	0	1	0	1	0	1	0	13	0	1	0	1	0	1	0	5				

The expansion from this position fails since we have actually reached the end of our string, so we end up copying the values again.

Idx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
S:	g	_	b	_	t	_	k	_	t	_	b	_	a	_	d	_	a	_	b	_	t	_	k	_	t	_	b
L:	1	0	1	0	1	0	5	0	1	0	1	0	1	0	13	0	1	0	1	0	1	0	5	0	1	0	1+

The position 26 becomes the next valid center, and the expansion from it fails due to the end of the string.

Idx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
S:	g	_	b	_	t	_	k	_	t	_	b	_	a	_	d	_	a	_	b	_	t	_	k	_	t	_	b
L:	1	0	1	0	1	0	5	0	1	0	1	0	1	0	13	0	1	0	1	0	1	0	5	0	1	0	1

Since we have swept through the entire L array, we now have the length of the longest palindromic substring that is centered at every possible position. The largest L value gives us the string-index of the longest palindromic substring (by taking the floor of division of the L-array-index by 2), which is "btktbadabtktb", centered on the letter 'd'.

Algorithm 8 lists the pseudocode for an implementation of Manacher's algorithm which does not require padding the string with placeholder characters for the "between letters" centers. The outer loop iterates over the indices of array L, and the inner loop maximally extends the current center's palindrome, but only starting with its already established current borders. If the palindrome has a side length (rather than being empty or just a single letter), then the algorithm starts copying over results from the left side into the right side, stopping as soon as a value is encountered which touches the boundary. The next index for the outer loop is then set to the index of that value, its current border is set to the copied value, and the whole process is repeated. The loops of the algorithm never visit the same character of S more than twice (once with the outer loop, and once potentially when copying its L value by the inner loop). Thus, it completes the task of finding the length of the longest palindrome at each possible center in just $O(|S|)$ time, requiring allocation of additional space only to hold the values of the L array.

Algorithm 8: manacher(S)

Input: A string S
Output: Array L of lengths of the longest palindromic substrings centered at each possible position

```
1  n = 2 * length(S) - 1 ;                    // |S| characters and |S| - 1 spaces between them
2  L = array of size n initialized to all zeroes
3  for (i = 0; i < n) do
4      strIdx = ⌊i/2⌋ ;                       // Obtain the character index within the string
5      d = ⌊L_i/2⌋ ;   // Get the length of the side of the so-far-longest palindrome known
        at this position (this may have been filled in by a previous palindrome's inner
        loop)
6      isSpace = i - 2 * strIdx ;   // 1 if i is odd and the center is a space, and 0 if it is
        even and the center is a character
       ;                              // Now extend the current center's palindrome maximally
7      while ((strIdx + d + 1 < |S|) and (strIdx - d - 1 + isSpace >= 0) and
        (S_{strIdx+d+1} == S_{strIdx-d-1+isSpace})) do
8          d++
9      L_i = 2 * d + 1 - isSpace ;                // And save its length into the L array
       ;  // If the length is longer than 1, then we have to copy L values from the left
        into the right side
10     if (L_i > 1) then
11         toCopy = 2 * d - isSpace
12         boundary = i + toCopy
13         for (j = 1; j <= toCopy) do
14             L_{i+j} = L_{i-j}
15             currentReach = L_{i+j} - 1
16             if (i + j + reach >= boundary) then
17                 L_{i+j} = boundary - (i + j) + 1 ;   // To make sure we're not going past the
                    boundary of the outer palindrome
18                 i = i + j - 1 ;   // Next outer loop iteration will continue from this center
19                 break
20 return L
```