


# Efficient Algorithms for Counting Gapped Palindromes

Andrei Popa ✉

Department of Computer Science, University of Bucharest, Romania

Alexandru Popa ✉ 

Department of Computer Science, University of Bucharest, Romania

---

## Abstract

A gapped palindrome is a string  $uvu^R$ , where  $u^R$  represents the reverse of string  $u$ . In this paper we show three efficient algorithms for counting the occurrences of gapped palindromes in a given string  $S$  of length  $N$ . First, we present a solution in  $O(N)$  time for counting all gapped palindromes without additional constraints. Then, in the case where the length of  $v$  is constrained to be in an interval  $[g, G]$ , we show an algorithm with running time  $O(N \log N)$ . Finally, we show an algorithm in  $O(N \log^2 N)$  time for a more general case where we count gapped palindromes  $uvu^R$ , where  $u^R$  starts at position  $i$  with  $g(i) \leq v \leq G(i)$ , for all positions  $i$ .

**2012 ACM Subject Classification** Theory of computation → Pattern matching

**Keywords and phrases** pattern matching, gapped palindromes, suffix tree

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2021.23

## 1 Introduction

A *gapped palindrome* is a string  $uvu^R$ , where  $u^R$  is the reverse of the string  $u$ . Gapped palindromes are a generalization of palindromes, which are strings of the form  $uu^R$ . In this paper we study the counting of occurrences of gapped palindromes in given string  $S$ .

Palindromes have many applications in bioinformatics, more precisely in the study of DNA and RNA sequences, where problems involving pairs of equal substrings and the repetition of certain substrings appear naturally and were intensively studied (see, for example, [20, 15, 11, 10, 14, 18, 12]). Manacher [13] presents an algorithm that finds for every position  $i$  of the string the longest palindrome centered in  $i$  in  $O(N)$  time. Using Manacher's algorithm we can count all occurrences of palindromes in linear time. We refer the reader to the book of Gusfield [7] and the references therein for more details regarding palindromes.

Gapped palindromes were first studied by Kolpakov and Kucherov in [9], where they design algorithms to find all maximal gapped palindromes in a string that obey two types of constraints termed long-armed and length-constrained palindromes. Dumitran, Gawrychowski and Manea present in [4] an algorithm which finds for each position  $i$  the longest string  $u$  such that  $uvu^R$  is a substring which has  $u^R$  starting at position  $i$  and  $g \leq |v| < G$ , where  $g$  and  $G$  are two given natural numbers. Another algorithm shown in the same paper solves a similar problem where  $|v|$  is only constrained by a lower bound function  $g(i)$ . Both algorithms have running time  $O(N)$ .

Brodal et al. [2] show an algorithm which finds all tuples  $(a, b, c, d)$  such that  $S[a..b] = S[c..d]$ ,  $c - b$  is in a given interval  $[g, G]$  and the pair of ranges  $[a..b]$  and  $[c..d]$  is maximal (it cannot be extended to the left or to the right such that the previous two properties still hold). The running time of Brodal et al.'s algorithm is  $O(N \log N + z)$ , where  $z$  is the number of the pairs found.



© Andrei Popa and Alexandru Popa;

licensed under Creative Commons License CC-BY 4.0

32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021).

Editors: Paweł Gawrychowski and Tatiana Starikovskaya; Article No. 23; pp. 23:1–23:13

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Counting palindromes was previously studied in [6] where it is shown that counting all distinct palindromes in a string can be done in linear time. Rubinchik and Shur describe in [16] an algorithm which counts distinct palindromes in a substring of a given string  $S$  of length  $N$  in time  $O(\log N)$ , with  $O(N \log N)$  preprocessing time.

In this paper we study counting occurrences of gapped palindromes without traversing each of them, the running time of our algorithms not depending on this number. We consider that the counting problem is a natural variation of the already studied related problems and we offer solutions for counting occurrences of gapped palindromes under various constraints. The first problem we solve is counting all strings of the form  $uvu^R$  in a given string  $S$  without any other constraints, for which we give a linear time complexity algorithm. For the second problem, which has the additional constraint that for  $g$  and  $G$  given natural numbers,  $g \leq |v| \leq G$ , we describe a solution in  $O(N \log N)$  time. Finally, we present the solution for the problem where for each position  $i$  of the given string  $S$  we find all substrings  $uvu^R$  where  $u^R$  begins at position  $i$  and  $g(i) \leq |v| \leq G(i)$ , for the given functions  $g$  and  $G$ , the algorithm having running time  $O(N \log^2 N)$ .

## 2 Preliminaries

A substring of a string  $S$  which starts at position  $i$  and ends at position  $j$  (inclusive) is denoted by  $S[i..j]$ . The length of the string  $S$  is  $|S|$ . We denote  $|S|$  by  $N$ . The  $i$ -th character of the string is denoted by  $S_i$ , the first character is  $S_1$  and the last is  $S_{|S|}$ . A string  $S$  which has the property that  $S = S^R$  is called a *palindrome*. We denote by *gapped palindrome* a string  $P = uvu^R$ , where  $u$  and  $v$  are arbitrary strings and  $|u| > 0$ . For  $|v| \leq 1$ ,  $P$  is also a palindrome. Note that a gapped palindrome  $P$  might be split into  $uvu^R$  in multiple ways. In this paper we consider occurrences of  $uvu^R$  and  $u'v'u'^R$  different if either  $u \neq u'$ ,  $v \neq v'$  or the positions at which the two substrings start are different.

A data structure used in all our algorithms is the *suffix tree* introduced by Weiner in [19], also described in Gusfield [7]. A suffix tree of a string  $S$  is a tree  $T$  where each edge  $(u, v)$  ( $u$  is the parent of  $v$ ) is labeled by a nonempty substring  $S[i..j]$  of  $S$ . In this tree, the concatenation of the labels of all edges on a path from the root to a certain node  $n$  forms a string which we denote by  $H(n)$ . A fundamental property of the suffix tree structure is that any two edges  $(u, v_1)$ ,  $(u, v_2)$  are labeled by strings which start with different characters. Thus, two different nodes  $n_1$  and  $n_2$  have the property that  $H(n_1) \neq H(n_2)$ . Moreover, it holds that  $\text{lcp}(H(n_1), H(n_2)) = H(\text{lca}(n_1, n_2))$ , where  $\text{lca}(n_1, n_2)$  represents the lowest common ancestor of the nodes  $n_1$  and  $n_2$  and  $\text{lcp}(H(n_1), H(n_2))$  is the longest common prefix of the strings  $H(n_1)$  and  $H(n_2)$ . For each suffix  $S[i..N]$  of  $S$  there is a node  $n$  such that  $S[i..N] = H(n)$  and by the previous property, the node  $n$  is unique, thus we denote it by  $T(i)$ . We term  $T(i)$  the *associated node* of the suffix  $S[i..N]$ . If  $S$  is of the form  $S'\$$  where  $\$$  is a character that does not appear in  $S$ , then the associated node of a suffix is a leaf. We denote the root of the tree by  $n_r$ ,  $M(n)$  is the substring which labels the edge from the parent of the node  $n$  to  $n$  (in particular  $M(n_r) = \epsilon$  where  $\epsilon$  is the empty string).

By  $\text{Anc}(n)$  we denote the set of ancestors of the node  $n$  (including  $n$ ) and by  $\text{Tree}(n)$  we denote the set of all nodes in the subtree of  $n$ . By the property of the  $\text{lcp}$  function we have that  $\text{lcp}(S[i..N], S[j..N]) = H(\text{lca}(T(i), T(j)))$ .

Assuming the alphabet of a string is of a constant size, the suffix tree can be constructed in  $O(N)$  time, as shown by Weiner in [19]. There is also an online algorithm to construct the suffix tree in linear time found by Ukkonen [17]. Given a suffix tree of a string, Gusfield [7] shows how to compute  $\text{lcp}(S[i..N], S[j..N])$  in  $O(1)$  time.

For a given string  $S$ , Manacher shows in [13] an algorithm which computes  $Pal_i$  representing the length of the longest palindrome centered in  $i$ , for all  $1 \leq i \leq N$ , in time complexity  $O(N)$ . By summing all these values we obtain an algorithm to count all occurrences of palindromes in  $S$  in linear time.

Harel and Tarjan [8] describe a method to partition a rooted tree  $T$  such that the nodes of each partition form a path and any two nodes in the same partition are one the ancestor of the other. An important property of the partitioning, is that a path from any node to the root intersects  $O(\log N)$  partitions. We refer to this kind of partitioning of a tree by *heavy path decomposition*.

An AVL Tree is a data structure introduced in [1]. An AVL tree is a balanced binary tree which holds a set of ordered elements. It can perform operations such as adding a new element to the set, removing an element, asking for the smallest element greater than a given value in  $O(\log N)$ , where  $N$  is the number of elements it holds. A property that we use in this paper is that two AVL trees of sizes  $N$  and  $M$  can be merged in  $O\left(\log\binom{N+M}{N}\right)$  time, as shown by Brown And Tarjan in [3].

In our algorithms we use *binary indexed trees* for computing and updating efficiently partial sums on an array where we have operations of the form “add  $x$  to the element on the position  $p$ ”. The binary indexed tree is described by Fenwick in [5]. For a binary indexed tree  $F$  associated to an array  $A$ , we denote the operation  $A[p] = A[p] + x$  by  $F[p] = F[p] + x$  and the query  $\sum_{i=1}^p A[i]$  by  $F(p)$ . Both operations are executed in running time  $O(\log N)$  where  $N$  is the length of the array.

We name *reversed binary indexed tree* a similar structure  $F'$  where  $F'(i) = \sum_{i=p}^N A[i]$ . The reversed binary indexed tree  $F'$  can be built using a standard binary indexed tree  $F$  where on an operation  $A[p] = A[p] + x$  we execute  $F[|A| - p + 1] = F[|A| - p + 1] + x$  and  $F'(i) = F(N - p + 1)$ .

### 3 Counting all gapped palindromes

In this section we study the following problem:

► **Problem 1** (Counting all occurrences of gapped palindromes). *Let  $S$  be a given string of length  $N$ . Count how many tuples  $(a, b, c, d)$  exist such that  $1 \leq a \leq b < c \leq d \leq N$  and  $S[a..b]^R = S[c..d]$ .*

*We denote the set of these tuples by  $Q$ .*

We begin with a fundamental observation.

► **Remark 2.** For fixed values  $c$  and  $b$ , the number of tuples  $(a, b, c, d)$  such that  $S[a..b]^R = S[c..d]$  is  $|lcp(S[1..b]^R, S[c..N])|$ .

**Proof.** Let  $L = |lcp(S[1..b]^R, S[c..N])|$ . For any  $l \in \{1..L\}$  we have that  $S[b-l+1..b]^R = S[c..c+l-1]$ , thus assigning  $a := b-l+1$  and  $d := c+l-1$  the property  $S[a..b]^R = S[c..d]$  holds. Also we have that  $S[b-l+1..b]^R \neq S[c..c+l-1]$  for any  $l > L$ , so for any assignment  $a := b-l+1$  and  $d := c+l-1$  the property  $S[a..b]^R = S[c..d]$  does not hold. For other values of  $a$  and  $d$ , the strings  $S[a..b]^R$  and  $S[c..d]$  have different length. Thus, the number of tuples  $(a, b, c, d)$  with the desired property is  $|\{1..L\}|$ , which, in turn, is equal to  $|lcp(S[1..b]^R, S[c..N])|$ . ◀

## 23:4 Counting Gapped Palindromes

Let  $T$  be the suffix tree built on the string  $S' = S\#S^R\$$  of length  $N'$ , where the characters  $\#$  and  $\$$  do not appear in the string  $S$ . We define  $j_r$  such that for  $j \in \{1..N\}$  we have  $S'[j..N] = S'[N + 2..j_r]^R$  and for  $j \in \{N + 2..2 \cdot N + 1\}$  we have  $S'[N + 2..j]^R = S'[j_r..N]$ . Thus  $j_r$  can be expressed as  $2 \cdot N + 2 - j$ .

► **Remark 3.** For some  $b$  and  $c$  such that  $1 \leq b, c \leq N$  we have that:

$$lcp(S[1..b]^R, S[c..N]) = lcp(S'[b_r..N'], S'[c..N'])$$

**Proof.** By the structure of  $S'$  we have that  $S'[b_r..N' - 1] = S[1..b]^R$  and  $S'[c..N] = S[c..N]$ . Let  $L$  be  $|lcp(S[1..b]^R, S[c..N])|$ . If  $L < \min(|S[1..b]^R|, |S[c..N]|)$  then  $S_{b-L} \neq S_{c+L}$ , so  $S'_{b_r+L} \neq S'_{c+L}$ , thus  $lcp(S'[b_r..N'], S'[c..N']) = L$ . If  $L = |S[1..b]^R|$  or  $L = |S[c..N]|$ , then either  $S'_{b_r+L} = \$$  or  $S'_{c+L} = \#$ . Because the characters  $\$$  and  $\#$  are unique in  $S'$  and  $b_r + L \neq c + L$ , we have that  $S'_{b_r+L} \neq S'_{c+L}$ , thus  $lcp(S'[b_r..N'], S'[c..N']) = L$ . ◀

The following remark indicates how to find pairs of a reversed prefix and a suffix which have a certain longest common prefix:

► **Remark 4.** For a prefix  $S[1..j]$  and a suffix  $S[i..N]$ , we have that  $lca(T(i), T(j_r)) = n$  is equivalent to the existence of two nodes  $n_1$  and  $n_2$  children of the node  $n$  such that  $T(i) \in Tree(n_1)$  and  $T(j_r) \in Tree(n_2)$ .

Using Remarks 3 and 4 we design Algorithm 1. Informally, Algorithm 1 performs as follows. For each node  $n$  of the suffix tree  $T$  built on the string  $S'$  we count the pairs of a reversed prefix and a suffix which have the  $lcp$  equal to  $H(n)$ , let this number be  $cnt$ . This number represents the number of pairs  $(b, c)$  such that  $lcp(S[1..b]^R, S[c..N]) = H(n)$  holds. We add to the final answer the value  $|H(n)| \times cnt$ , where  $|H(n)|$  comes from the number of pairs  $(a, d)$  such that  $S[a..b]^R = S[c..d]$  for each of the  $(b, c)$  pairs. To count the number of pairs  $(b, c)$ , we iterate over the children of  $n$  and we count the number of pairs formed by reversed prefixes and suffixes in the subtree of the current child with suffixes and reversed prefixes in the subtrees of the previous children.

► **Lemma 5.** The Algorithm 1 gives the value  $ans_1 = \sum_{b=1}^N \sum_{c=1}^N lcp(S[1..b]^R, S[c..N])$  in  $O(N)$  time.

**Proof.** Remarks 3 and 4 prove that the algorithm correctly counts for each node  $n$  the number of pairs  $(b, c)$  such that  $lcp(S[1..b]^R, S[c..N]) = H(n)$ . Every pair  $(b, c)$  is processed when  $n = lca(T(b_r), T(c))$ , adding the value  $H(n)$  to  $ans_1$ , thus the value  $ans_1$  is  $\sum_{b=1}^N \sum_{c=1}^N lcp(S[1..b]^R, S[c..N])$ . The time complexity of the algorithm is linear since its core is a simple *depth-first* traversal (with some constant time additional computations). ◀

Algorithm 1 does not count the elements of the answer set  $Q$  because the condition  $b < c$  does not hold. Nevertheless, we can assert the following:

► **Remark 6.** Let

$$Q' = \{(a, b, c, d) \mid 1 \leq a \leq b \leq N, 1 \leq c \leq d \leq N, S[a..b]^R = S[c..d]\}$$

Then  $ans_1 = |Q'|$ .

**Proof.** It follows immediately from Lemma 5 and Remark 2. ◀

We observe that  $Q \subset Q'$ . We aim to find  $|Q|$  by computing  $|Q'|$  and  $|Q' - Q|$ .

■ **Algorithm 1** Counting excluding the condition  $b < c$ .

---

```

1 Procedure iterate( $n$ : the current node of the suffix tree)
   Data:  $Sf[u]$  - the number of the suffixes of  $S$  in the subtree of the node  $u$ 
          $Pr[u]$  - the number of the prefixes of  $S$  in the subtree of the node  $u$ 
2    $pr \leftarrow 0$ ;
3    $sf \leftarrow 0$ ;
4    $cnt \leftarrow 0$ ;
5   if  $n$  is leaf then
6      $i \leftarrow T^{-1}(n)$ ;           //  $n$  is the associated node of a suffix of  $S'$ 
7     if  $i \leq N$  then                //  $i$  corresponds to a suffix of  $S$ 
8        $sf \leftarrow 1$ ;
9     else if  $i > N + 1$  and  $i \leq 2 \cdot N + 1$  then //  $i_r$  corresponds to a prefix
        of  $S$ 
10       $pr \leftarrow 1$ ;
11    end
12  else
13    for  $ch \leftarrow$  child of  $n$  do
14      iterate( $ch$ );
15       $cnt \leftarrow cnt + Sf[ch] \cdot pr$ ;
16       $cnt \leftarrow cnt + Pr[ch] \cdot sf$ ;
17       $pr \leftarrow pr + Pr[ch]$ ;
18       $sf \leftarrow sf + Sf[ch]$ ;
19    end
20  end
21   $ans_1 \leftarrow cnt \times |H(n)|$ ;
22   $Sf[u] \leftarrow sf$ ;
23   $Pr[u] \leftarrow pr$ ;
24 iterate( $n_r$ ) // start the iteration from the root of the suffix tree

```

---

► **Remark 7.** Let  $(a, b, c, d)$  be a tuple in  $Q$ . Then  $(a, b, c, d) \in Q'$  but also  $(c, d, a, b) \in Q'$  and  $(a, b, c, d) \neq (c, d, a, b)$ . Moreover, if  $(a, b, c, d) \in Q'$  and  $[a, b] \cap [c, d] = \emptyset$ , then  $(a, b, c, d) \in Q$  or  $(c, d, a, b) \in Q$ .

In other words, a tuple  $(a, b, c, d) \in Q$  “appears twice” in  $Q'$  and all the elements in  $Q'$  which have this property are those for which the substrings  $S[a..b]$  and  $S[c..d]$  do not intersect (either  $b < c$  or  $d < a$ ). We use Remark 7 to compute the final answer using  $ans_1$  and the number of tuples  $(a, b, c, d)$  such that  $S[a..b]$  and  $S[c..d]$  do not intersect.

► **Lemma 8.** Let  $I = \{(a, b, c, d) \in Q' \mid [a, b] \cap [c, d] \neq \emptyset\}$  and  $ans_2 = |I|$ . Then the answer to the Problem 1 is  $ans = (ans_1 - ans_2)/2$ .

**Proof.** From Remark 7 we have that  $|Q| = |Q' \setminus I|/2$ , and as  $I \subset Q'$ , we have  $|Q' \setminus I| = |Q'| - |I|$ . ◀

We say that a palindrome  $S[a..b]$  is *centered* at  $(a + b)/2$  and the value  $(a + b)/2$  is the center of the palindrome.

► **Lemma 9.** Let  $(a, b, c, d) \in I$ . Then  $S[a..b]$  and  $S[c..d]$  are contained in a palindrome centered at  $(b + c)/2$ . More precisely, there are  $l$  and  $r$  such that  $l \leq a, b, c, d \leq r$  and  $S[l..r]$  is a palindrome and  $(l + r)/2 = (b + c)/2$ .

## 23:6 Counting Gapped Palindromes

**Proof.** Assume that  $b \leq d$ . The case  $b > d$  is solved similarly.

As  $S[a..b]$  and  $S[c..d]$  intersect, we have that  $b \geq c$ . Thus  $S$  has the following structure:

$$S = XS[a..c-1]S[c..b]S[b+1..d]Y$$

Due to the fact that  $S[a..b]^R = S[c..d]$  and  $b \leq d$  we have that  $S[a..c-1]S[c..b]^R = S[c..b]S[b+1..d]$ , thus  $S[c..b]^R = S[c..b]$  and  $S[a..c-1]^R = S[b+1..d]$ . Therefore  $S$  has the structure  $XUAU^RY$ , where  $A$  is a palindrome centered at  $(b+c)/2$ . It follows that  $S[a..d] = UAU^R$  also has the center  $(b+c)/2$ . ◀

To count  $(a, b, c, d) \in I$ , we compute for each  $c_e \in [1, N]$ ,  $2 \cdot c_e \in \mathbb{N}$  the value  $V_{c_e}$  representing the number of tuples which are included in a palindrome centered at  $c_e$  and  $c_e = (b+c)/2$ . Then  $ans_2 = \sum_{c_e} V_{c_e}$ .

► **Lemma 10.** Let  $S[l..r]$  be the longest palindrome centered at  $c_e$ . Then  $V_{c_e} = \lceil (r-l+1)/2 \rceil^2$ .

**Proof.** Observe that  $a$  can take any integer value in the interval  $[l, \lfloor (l+r)/2 \rfloor]$ ,  $b$  can take any integer value in  $\lceil (l+r)/2 \rceil, r$ . By Lemma 9, the values  $c$  and  $d$  are the symmetric of  $b$  and  $a$  with respect to  $c_e$ . More precisely  $c = (l+r) - b$  and  $d = (l+r) - a$ . Thus the number of tuples  $(a, b, c, d)$  is  $(\lfloor (l+r)/2 \rfloor - l + 1) \cdot (r - (\lceil (l+r)/2 \rceil) + 1)$  which is equal to  $\lceil (r-l+1)/2 \rceil^2$ . ◀

We use all the previous results in designing Algorithm 2. We can thus state the theorem:

► **Theorem 11.** Problem 1 can be solved in  $O(N)$  time.

**Proof.** Algorithm 2 computes the desired value  $ans = |Q|$ . The correctness of Algorithm 1 is given by Lemma 5. The step 4 is correct by Lemma 10. Lemma 8 proves the correctness of the step 6.

The suffix tree of the string  $S'$  in step 1 is computed in  $O(N)$  time using Ukkonen's algorithm [17]. Algorithm 1 runs in linear time by Lemma 5. The lengths of the longest palindromes centered in each position are computed using Manacher's algorithm [13] in  $O(N)$ . Step 5 is done in linear time and Step 6 takes constant time. Thus the entire algorithm runs in  $O(N)$  time. ◀

■ **Algorithm 2** The algorithm for solving Problem 1.

- 
- 1 build *suffix tree* on the string  $S' = S\#S^R\$$  ;
  - 2 compute  $ans_1$  using Algorithm 1 ;
  - 3 for each center  $c$  compute the longest palindrome in  $S$  centered at  $c$  ;
  - 4 compute the values  $V_c = \lceil (r-l+1)/2 \rceil^2$  where  $S[l..r]$  is the longest palindrome centered at  $c$  ;
  - 5 compute  $ans_2$  as  $\sum_{c=2}^{2 \cdot N} V_{c/2}$  ;
  - 6 compute  $ans = (ans_1 - ans_2)/2$  ;
-

#### 4 Palindromes with constraints on the length of the gap

In this section we consider a version of Problem 1 with the additional constraint that  $Q$  contains only tuples with the property that  $g \leq c - b - 1 \leq G$ , for given  $g$  and  $G$  positive integers.

► **Problem 12** (Counting palindromes with gap length constraint). *Let  $S$  be a string of length  $N$  and  $g, G$  positive integers. Let*

$$Q = \{(a, b, c, d) \mid 1 \leq a \leq b < c \leq d \leq N, g \leq c - b - 1 \leq G, S[a..b]^R = S[c..d]\}$$

. Find  $|Q|$ .

We use an adaptation of Algorithm 1. Instead of counting for each node  $n$  all the pairs  $(i, j)$  such that  $\text{lcp}(S[1..j]^R, S[i..N]) = H(n)$ , we count only those that also hold the constraint on the gap, namely  $g \leq i - j - 1 \leq G$ . This approach obtains the correct answer directly without the need of a strategy to subtract wrongly counted tuples as in the Problem 1. However, a data structure on an ordered set is needed, thus increasing the running time of the algorithm.

Brodal et al. [2] describe an algorithm that solves a similar problem. For a node  $n$  of the suffix tree, Brodal et al.'s algorithm processes pairs of suffixes belonging to the subtrees of two different children. For a suffix  $S[i..N]$  in one subtree the algorithm finds the smallest index  $j$  of a suffix  $S[j..N]$  in the other subtree, such that  $j \geq i + v$ . The value  $v$  is independent of  $i$  and  $j$ , but depends on the node  $n$ . The algorithm also iterates on some interval of suffixes starting from  $j$ , thus its time complexity depends on the number of suffixes individually found. Because in our problem we need only to find the number of the elements in an interval, we change the algorithm to have a running time depending only on  $N$ .

The modification of the strategy presented in Brodal et al. [2] needs the following lemma:

► **Lemma 13.** *Let  $E$  be a list of sorted elements and  $T$  an AVL tree of sizes  $N$  and  $M$  respectively, such that  $N \leq M$ . For each element  $i$  of  $E$  we can find the biggest element  $j$  in  $T$  such that  $j \leq i$  and its position in  $T$  (in other words, how many elements of  $T$  are smaller than  $j$ ) in time  $O\left(\log\binom{N+M}{N}\right)$ .*

**Proof.** From Lemma 3 of Brodal et al. [2] we know that for each  $i$  in  $E$  we can find the smallest  $j$  in  $T$  such that  $j \geq i$  in time  $O\left(\log\binom{N+M}{N}\right)$ . In our algorithm we use a similar operation.

Now we describe the method of finding the position of the element  $j$ . We denote by  $L(n)$  the left child of the node  $n$  and by  $R(n)$  the right child of the node  $n$ . For each node  $u$  from the AVL we keep a table  $\text{Size}(u)$  representing the size of the subtree rooted in the node  $u$ . When inserting a new element in the AVL tree, we update the value  $\text{Size}(u)$  to be  $\text{Size}(L(u)) + \text{Size}(R(u)) + 1$  at the step the node  $u$  is visited. For finding the position of an element, during the traversal of the tree, we keep a variable  $p$  which counts the number of smaller elements than the element in the current node which are not situated in the current subtree. When visiting the left child, we leave  $p$  unchanged, when visiting the right child we add to  $p$  the value  $\text{Size}(L(u)) + 1$ , and when returning to the parent we undo the change made when we visited the node  $u$ .

The running time of the algorithm does not change, thus the operation can be performed in  $O\left(\log\binom{N+M}{N}\right)$ . ◀



► **Theorem 14.** *Problem 12 can be solved in running time  $O(N \log N)$ .*

**Proof.** We apply the strategy described by Brodal et al. in [2]. We change the suffix tree to have each node with at most two children in order to use “the smaller half trick”. For each node  $n$  we maintain an AVL tree that stores all the suffixes  $i$  in the subtree of  $n$  and another AVL tree that stores all the prefixes  $j$  in the subtree of  $n$ . The AVL tree for a node  $n$  is computed as the union of the AVL trees of its children, which can be done in time  $O\left(\log \binom{s_1+s_2}{s_1}\right)$  where  $s_1$  and  $s_2$  are the number of nodes in the two subtrees of node  $n$ . All union operations are processed in  $O(N \log N)$  time.

For a node  $n$  of the suffix tree, let  $n_1$  be the child with the smaller subtree and  $n_2$  the child with the bigger subtree. For each suffix  $i$  of  $S$  in the subtree of  $n_1$  we find how many prefixes  $j$  of  $S$  in the subtree of  $n_2$  exist such that  $g \leq i - j - 1 \leq G$ . We also find for each prefix  $j$  in the subtree of  $n_1$  the number of suffixes  $i$  in the subtree of  $n_2$  such that  $g \leq i - j - 1 \leq G$ . We add both numbers to the answer.

We describe the first case, the other one being similar. For each suffix  $i$  in the subtree of  $n_1$  count the number of prefixes  $j_1$  in the subtree of  $n_2$  that hold the condition  $g \leq i - j_1 - 1$  and the number of prefixes  $j_2$  which hold the condition  $G + 1 \leq i - j_2 - 1$ , the result being the difference of the two numbers. We rewrite  $g \leq i - j_1 - 1$  as  $j \leq i - g - 1$ . We merge the sorted list of the suffixes of  $n_1$  with the AVL tree which holds the prefixes of the subtree rooted at  $n_2$ . The elements  $i$  from the sorted list are treated as they would be  $i - g - 1$  and we return the largest  $j$  in the AVL which is smaller than or equal to  $i - g - 1$  and its position. If the position of  $j$  is  $p_j$ , then the number of prefixes counted for the suffix  $i$  is  $p_j$ .

The running time of our algorithm is  $O(N \log N)$ , being just a modification of the algorithm described by Brodal et al. in [2]. ◀

## 5 Gapped palindromes with positional constraints on the length of the gap

► **Problem 15** (Counting gapped palindromes for every position). *Let  $S$  be a string of size  $N$ . We say that for an index  $i$  a pair  $(j, r)$  is valid if  $j < i$  and  $S[i..i + r - 1]^R = S[j - r + 1..j]$ . Given two functions  $g$  and  $G$ , count for each index  $i$  the number  $P_i$  representing the number of valid pairs  $(j, r)$  for which the property  $g(i) \leq i - j - 1 \leq G(i)$  holds.*

We begin by reducing the problem of counting pairs  $(j, r)$  where  $j$  is bounded from both sides to one where  $j$  is only upper bounded.

► **Remark 16.** Let  $P_i^g$  be the number of valid pairs  $(j, r)$  of  $i$  such that  $i - j - 1 \geq g(i)$  and  $P_i^G$  the number of valid pairs  $(j, r)$  of  $i$  such that  $i - j - 1 \geq G(i) + 1$  (equivalently  $j \leq i - g(i) - 1$  and  $j \leq i - G(i) - 2$  respectively). Then  $P_i = P_i^g - P_i^G$ .

The following remark shows how to compute the values  $P_i^g$  and  $P_i^G$  based on Remark 2:

► **Remark 17.** The values  $P_i^g$  and  $P_i^G$  can be computed in the following way:

$$P_i^g = \sum_{j=1}^{i-g(i)-1} |\text{lcp}(S[1..j]^R, S[i..N])|$$

$$P_i^G = \sum_{j=1}^{i-G(i)-2} |\text{lcp}(S[1..j]^R, S[i..N])|$$



We provide a data structure to compute the value  $\sum_{j=1}^K |lcp(S[1..j]^R, S[i..N])|$  where initially  $K = 0$  and we make updates of the form  $K = K + 1$ . We denote by  $Q(i) = \sum_{j=1}^K lcp(S[1..j]^R, S[i..N])$  at a certain state given by  $K$ .

Let  $W$  be an array indexed by the set of the nodes of  $T$ .

Initially we have that  $W[n] = 0, \forall n \in T$ . On an update  $K = K + 1$  we execute  $W[u] = W[u] + |M(u)|$  for each  $u$  ancestor of  $T((K + 1)_r)$  (recall that  $(K + 1)_r = 2 \cdot N + 2 - (K + 1)$ ).

► **Remark 18.** Let  $A$  be the set of prefixes processed at a certain moment, more precisely  $A = \{T(j_r) \mid j \leq K\}$ ,  $K$  is the index of the last prefix added to the structure. Then  $W[n] = |M(n)| \cdot |A \cap Tree(n)|$ .

We can find the value  $Q(i)$  using the following lemma.

► **Lemma 19.**

$$Q(i) = \sum_{u \in Anc(T(i))} W[u]$$

**Proof.** By Remark 18 we have that

$$\begin{aligned} \sum_{u \in Anc(T(i))} W[u] &= \sum_{u \in Anc(T(i))} |M(u)| \cdot |A \cap Tree(u)| \\ &= \sum_{u \in Anc(T(i))} \sum_{v \in A \cap Tree(u)} |M(u)| \\ &= \sum_{\substack{(u,v) \in Anc(T(i)) \times A \\ v \in Tree(u)}} |M(u)| \\ &= \sum_{\substack{(u,v) \in Anc(T(i)) \times A \\ u \in Anc(v)}} |M(u)| \\ &= \sum_{v \in A} \sum_{u \in Anc(T(i)) \cap Anc(v)} |M(u)| \\ &= \sum_{v \in A} \sum_{u \in Anc(lca(T(i), v))} |M(u)| \end{aligned}$$

but  $A = \{T(j_r) \mid j \leq K\}$ , thus

$$\begin{aligned} &= \sum_{j \leq K} \sum_{u \in Anc(lca(T(i), T(j_r)))} |M(u)| \\ &= \sum_{j \leq K} |H(lca(T(i), T(j_r)))| \\ &= Q(i) \end{aligned}$$

We describe a method to update and compute the sum over the array  $W$ . Build the *heavy path decomposition* of the suffix tree  $T$ . Let  $Ch(n)$  be the partition corresponding to the node  $n$ . By looking at the partitions of the decomposition as sequences, we define  $Pos(n)$  being the position of  $n$  in the sequence  $Ch(n)$  and  $L_l(p)$  the node at position  $p$  in the partition  $l$ . We denote by  $Par(n)$  the parent of the node  $n$  and  $Par(n_r) = nil$  where  $nil$  denotes the absence of a value. The partition has the following property:

## 23:10 Counting Gapped Palindromes

► **Property 20.**  $Par(L_l(i)) = L_l(i - 1)$  for each  $i > 1$ . Moreover, if  $u$  is an ancestor of  $v$ , then  $L_{Ch(u)}(i)$  is also an ancestor of  $v$ , for all  $i \leq Pos(u)$ .

Harel and Tarjan show in [8] a property of the heavy path decomposition which we reformulate here:

► **Property 21.** The number of distinct partitions encountered on a path from a node  $n$  to the root is  $O(\log N)$ . In other words, consider an algorithm which has a variable  $n$  initially equal to some node of the tree. The number of steps in a loop of the form “while  $n \neq nil$  do  $l = Ch(n)$ ,  $n = Par(L_l(1))$ ” is of the order  $O(\log N)$ .

By Properties 20 and 21 we design Algorithms 3 and 4, which update the structure when executing  $K = K + 1$  and query the structure respectively.

■ **Algorithm 3** The update procedure.

---

```

1 Procedure update()
  Data:  $St_l[j]$  - for each  $l$  an array with the values  $W[u]$  corresponding to the
        chain  $l$ , meaning  $St_l[j] = W[L_l(j)]$ 
2   $n \leftarrow T((K + 1)_r)$ ; // the leaf corresponding to the added prefix
3   $K := K + 1$ ;
4  while  $n \neq nil$  do
5     $p \leftarrow Pos(n)$ ;
6     $l \leftarrow Ch(n)$ ;
7    for  $j \leftarrow 1..p$  do // operation executed by an efficient structure
8       $St_l[j] = St_l[j] + |M(L_l(j))|$ ; // add  $|M(u)|$  on the chain
9    end
10    $n \leftarrow Par(L_l(1))$ ;
11 end

```

---

■ **Algorithm 4** The query function.

---

```

1 Function query( $i$ : the index of the suffix)
  Data:  $St_l[j]$  - for each  $l$  the array with the values  $W[u]$  corresponding  $l$  to the
        chain  $l$ , meaning  $St_l[j] = W[L_l(j)]$ 
2   $n \leftarrow T(i)$ ; // the leaf corresponding to the added suffix
3   $sum \leftarrow 0$ ;
4  while  $n \neq nil$  do
5     $p \leftarrow Pos(n)$ ;
6     $l \leftarrow Ch(n)$ ;
7    for  $j \leftarrow 1..p$  do // operation executed by an efficient structure
8       $sum = sum + St_l(j)$ 
9    end
10    $n \leftarrow Par(L_l(1))$ ;
11 end
12 return  $sum$ 

```

---

► **Lemma 22.** Algorithms 3 and 4 simulate correctly the operations on  $W$ . The time complexity of Algorithm 3 is  $O(\log N)$  operations of the form “assign  $B[i] = B[i] + A[i]$  for all  $i \leq p$ ”, where  $p$  is a given integer. Algorithm 4 runs in  $O(\log N)$  time operations of the form “compute  $\sum_{i=1}^p B[i]$ ” for a given  $p$ .

**Proof.** We prove only the statements about Algorithm 3, those concerning Algorithm 4 can be proven similarly.

Consider line 7 of Algorithm 3. If we reverse the direction of the loop, then, by Property 20, the nodes  $L_l(j)$  on the next line provide the iteration of the ancestors of the node in order, from  $n$  to the chain above. Line 10 takes the next node on the path from  $n$  to the root which is processed at the next iteration of the exterior loop (when  $j$  on line 7 is equal to  $p$ ). Thus, every ancestor of  $n$  is eventually visited.

The time complexity of the algorithm is a direct consequence of Property 21, because in each iteration of the exterior loop (line 4 of Algorithm 3) an operation of the form  $St_l[j] = St_l[j] + |M(L_l(j))|$  is executed. ◀

Finally we provide a data structure to perform the operations on the arrays  $St_l$  in Algorithms 3 and 4. More precisely, we solve the following problem:

► **Problem 23.** Let  $A_i$  be a sequence and  $B$  be an array, both of size  $N$ , initially  $B[i] = 0, \forall i$ . Execute a series of operations of the form:

- update: given  $p$ , assign  $B[i] = B[i] + A_i$  for all  $i \leq p$
- query: given  $p$ , find  $\sum_{i=1}^p B[i]$

► **Lemma 24.** Problem 23 can be solved by executing each operation in running time  $O(\log N)$ .

**Proof.** Let  $S_i = \sum_{j=1}^i A_j$ , be the partial sum sequence of  $A$  which can be computed in linear time and  $P_i$  the sequence of values  $p$  which were used to update the values of  $B$  up to the current step. Let  $L$  be the length of the sequence  $P$ . For a given position  $p$  we can compute the partial sum query in the following way:

$$\sum_{i=1}^p B[i] = \sum_{j=1}^L \sum_{i=1}^p A_i^j$$

where  $A_i^j = A_i$  for  $i \leq P_j$  and  $A_i^j = 0$  otherwise. Then, let  $P^{\leq p}$  be the sequence with values of  $P$  such that  $P_i \leq p$  and  $P^{>p}$  the sequence with values  $P_i > p$ , with lengths  $L^{\leq p}$  and  $L^{>p}$  respectively. We have:

$$\begin{aligned} \sum_{i=1}^p B[i] &= \sum_{j=1}^{L^{\leq p}} S[P_j^{\leq p}] + \sum_{j=1}^{L^{>p}} S[p] \\ \sum_{i=1}^p B[i] &= \left( \sum_{j=1}^{L^{\leq p}} S[P_j^{\leq p}] \right) + S[p] \cdot L^{>p} \end{aligned}$$

or, for a better view:

$$\sum_{i=1}^p B[i] = \sum_{j=1}^{L^{\leq p}} S[P_j^{\leq p}] + S[p] \cdot \sum_{j=1}^{L^{>p}} 1$$

Thus, we separate the computation of  $\sum_{i=1}^p B[i]$  in two, easier to compute, sums. The first sum,  $\sum_{j=1}^{L^{\leq p}} S[P_j^{\leq p}]$  can be computed using a binary indexed tree  $F_1$ , which for a given  $p$  we update by executing  $F_1[p] = F_1[p] + S_p$ . We compute the second sum,  $S[p] \cdot \sum_{j=1}^{L^{>p}} 1$  by using a reversed binary indexed tree  $F_2$ , which for a given  $p$  we update by executing  $F_2[p] = F_2[p] + 1$ . The answer for a query of the form  $\sum_{i=1}^p B[i]$  is  $F_1(p) + S_p \cdot F_2(p+1)$ . ◀

## 23:12 Counting Gapped Palindromes

All the above lemmas and remarks are joint together in Algorithm 5.

■ **Algorithm 5** The algorithm which solves Problem 15.

---

```

1 build the suffix tree of the string  $S' = S\#S^R\$$ ;
2 compute the pairs  $(i, i - g(i) - 1)$  and  $(i, i - G(i) - 2)$  and sort them by the second
  field;
3 for  $K \leftarrow 0$  to  $N - 1$  do
4   for  $(i, \_)$   $\leftarrow$  remaining pairs which have the second field equal to K do
5      $P_i^g$  or  $P_i^G := \text{query}(i)$ ;
6   end
7   update();
8 end
9 for  $i \leftarrow 1$  to  $N$  do
10   $P_i = P_i^g - P_i^G$ ;
11 end

```

---

We conclude with the following theorem:

► **Theorem 25.** *Problem 15 can be solved in time complexity  $O(N \log^2 N)$ .*

**Proof.** By Lemma 22 we have that Algorithms 3 and 4 correctly simulate the operations on the array  $W$  and the Lemma 19 shows that the operations on the array  $W$  correctly computes the value  $Q(i)$ . Remark 17 shows that Algorithm 5 correctly computes  $P^g(i)$  and  $P^G(i)$ , then computes  $P_i$  using Remark 16. Thus, the correctness of the algorithm is proven.

Using Lemma 24 we obtain that Algorithms 3 and 4 have the total time complexity  $O(\log^2 N)$ . Algorithm 5 does  $N$  calls of Algorithm 3. The query function is called two times for each position  $i$  (once for  $P^g(i)$  and once for  $P^G(i)$ ), thus there are  $2 \cdot N$  calls of Algorithm 4. Therefore the total running time of the algorithm is  $O(N \log^2 N)$ . ◀

## 6 Conclusions and future work

In this paper we show a linear time algorithm that counts the number of gapped palindromes in a string. Then, we show an algorithm with time complexity  $O(N \log N)$  for a variation of the problem in which we have a lower and an upper bound on the length of the gap of the palindromes counted. Finally, we show an algorithm with  $O(N \log^2 N)$  time complexity for a more general case where we count gapped palindromes  $uvu^R$ , where  $u^R$  starts at position  $i$  with  $g(i) \leq v \leq G(i)$ , for all positions  $i$ .

As an open problem, we believe that it is possible to solve Problem 15 using an algorithm that has  $O(N \log N)$  running time. One possible research direction is to adapt the algorithm in Section 4. However, we mention that a straightforward adaptation is not possible for the following reason. In our suffix tree traversal, we count for every prefix/suffix from the smallest subtree the corresponding pair from the other subtree, while the goal is to count for every suffix the corresponding prefixes irrespective of the subtree they belong to.

Another approach to obtain a  $O(N \log N)$  algorithm for Problem 15 is to use a suffix array instead of a suffix tree. Then, we tried to use the fact that the length of the longest common prefix of two suffixes from the suffix array is the minimum length of the longest common prefix of all the pairs of consecutive suffixes between the two given suffixes. The difficulty consists in building a data structure which handles the change of the minimum longest common prefix of two consecutive suffixes in an interval and queries sums on partial minimum longest common prefix of two consecutive suffixes both in time complexity  $O(\log N)$ .

---

**References**


---

- 1 Georgy Adelson-Velsky and Evgenii Landis. An algorithm for organization of information. *Doklady Akademii Nauk SSSR*, 146(2):263–266, 1962.
- 2 Gerth Stølting Brodal, Rune B Lyngsø, Christian NS Pedersen, and Jens Stoye. Finding maximal pairs with bounded gap. In *Annual Symposium on Combinatorial Pattern Matching*, pages 134–149. Springer, 1999.
- 3 Mark R Brown and Robert E Tarjan. A fast merging algorithm. *Journal of the ACM (JACM)*, 26(2):211–226, 1979.
- 4 Marius Dumitran, Paweł Gawrychowski, and Florin Manea. Longest Gapped Repeats and Palindromes. *Discrete Mathematics & Theoretical Computer Science*, Vol. 19 no. 4, FCT '15, 2017. doi:10.23638/DMTCS-19-4-4.
- 5 Peter M Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and experience*, 24(3):327–336, 1994.
- 6 Richard Groult, Élise Prieur, and Gwénaél Richomme. Counting distinct palindromes in a word in linear time. *Information Processing Letters*, 110(20):908–912, 2010.
- 7 Dan Gusfield. Algorithms on stings, trees, and sequences: Computer science and computational biology. *Acm Sigact News*, 28(4):41–60, 1997.
- 8 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *siam Journal on Computing*, 13(2):338–355, 1984.
- 9 Roman Kolpakov and Gregory Kucherov. Searching for gapped palindromes. *Theoretical Computer Science*, 410(51):5365–5373, 2009.
- 10 Tomoko Kuroda-Kawaguchi, Helen Skaletsky, Laura G Brown, Patrick J Minx, Holland S Cordum, Robert H Waterston, Richard K Wilson, Sherman Silber, Robert Oates, Steve Rozen, et al. The azfc region of the y chromosome features massive palindromes and uniform recurrent deletions in infertile men. *Nature genetics*, 29(3):279–286, 2001.
- 11 David RF Leach. Long dna palindromes, cruciform structures, genetic instability and secondary structure repair. *Bioessays*, 16(12):893–900, 1994.
- 12 Le Lu, Hui Jia, Peter Dröge, and Jinming Li. The human genome-wide distribution of dna palindromes. *Functional & integrative genomics*, 7(3):221–227, 2007.
- 13 Glenn Manacher. A new linear-time“on-line”algorithm for finding the smallest initial palindrome of a string. *Journal of the ACM (JACM)*, 22(3):346–351, 1975.
- 14 Sjoerd Repping, Helen Skaletsky, Julian Lange, Sherman Silber, Fulco van der Veen, Robert D Oates, David C Page, and Steve Rozen. Recombination between palindromes p5 and p1 on the human y chromosome causes massive deletions and spermatogenic failure. *The American Journal of Human Genetics*, 71(4):906–922, 2002.
- 15 Steve Rozen, Helen Skaletsky, Janet D Marszalek, Patrick J Minx, Holland S Cordum, Robert H Waterston, Richard K Wilson, and David C Page. Abundant gene conversion between arms of palindromes in human and ape y chromosomes. *Nature*, 423(6942):873–876, 2003.
- 16 Mikhail Rubinchik and Arseny M Shur. Counting palindromes in substrings. In *International Symposium on String Processing and Information Retrieval*, pages 290–303. Springer, 2017.
- 17 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
- 18 Peter E Warburton, Joti Giordano, Fanny Cheung, Yefgeniy Gelfand, and Gary Benson. Inverted repeat structure of the human genome: the x-chromosome contains a preponderance of large, highly homologous inverted repeats that contain testes genes. *Genome research*, 14(10a):1861–1869, 2004.
- 19 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11. IEEE, 1973.
- 20 DA Wilson and CA Thomas Jr. Palindromes in chromosomes. *Journal of molecular biology*, 84(1):115–138, 1974.