A Subquadratic Algorithm for
Minimum Palindromic Factorization

Gabriele Fici\textsuperscript{b}, Travis Gagie\textsuperscript{a}, Juha Kärkkäinen\textsuperscript{a}, Dominik Kempa\textsuperscript{a}

\textsuperscript{a}Department of Computer Science, University of Helsinki, Finland
\textsuperscript{b}Dipartimento di Matematica e Informatica, Università di Palermo, Italy

Abstract

We give an $O(n \log n)$ time algorithm for factoring a string into the minimum number of palindromic substrings. That is, given a string $S[1..n]$, in $O(n \log n)$ time our algorithm returns the minimum number of palindromes $S_1, \ldots, S_\ell$ such that $S = S_1 \cdots S_\ell$.

Keywords: string algorithms, palindromes, factorization

1. Introduction

Palindromic substrings are a well-studied topic in stringology and combinatorics on words. Since a single character is a palindrome, there are always between $n$ and $\left(\frac{n}{2}\right) + n = \Theta(n^2)$ non-empty palindromic substrings in a string of length $n$. There are only $2n - 1$ possible centers of those substrings, however — i.e., the $n$ individual characters and the $n - 1$ gaps between them — so many algorithms involving palindromic substrings still run in subquadratic time. For example, Manacher \cite{manacher} gave a linear-time algorithm for listing all the palindromic prefixes of a string. Apostolico, Breslauer and Galil \cite{abg} observed that Manacher’s algorithm can be used to list in linear time all maximal palindromic substrings, which are those that cannot be extended without changing the position of the center. Other linear-time algorithms for this problem were given by Jeuring \cite{jeuring} and Gusfield \cite{gusfield}. Since any palindromic substring is contained within the maximal palindromic substring with the same center, the list of all maximal palindromic substrings can be viewed as a linear-space representation of all palindromic substrings. For more discussion of algorithms involving palindromes, we refer the reader to Jeuring’s recent survey \cite{jeuring}.

Palindromes are a useful tool for investigating string complexity; see, e.g., \cite{fici}. A natural measure of the asymmetry of a string $S$ is its palindromic length $\text{PL}(S)$, which is the minimum number of palindromic substrings into which $S$...
can be factored. That is, \( PL(S) \) is the minimum number \( \ell \) such that there exist palindromes \( S_1, \ldots, S_{\ell} \) whose concatenation \( S_1 \cdots S_{\ell} = S \). For example, \( PL(abaab) = 2 \) and \( PL(abaca) = 3 \). Notice that, since a single character is a palindrome, \( PL(S) \) is always well-defined and lies between 0 and \( |S| \).

We became interested in palindromic length because of a recent conjecture by Frid, Puzynina and Zamboni \[5\]. Some infinite strings (e.g., the regular paperfolding sequence) are highly asymmetric in that they contain only a finite number of distinct palindromic substrings; see \[4\] for more discussion. For such strings, the palindromic length of any finite substring is proportional to that substring’s length. In contrast, for other infinite strings (e.g., the infinite power of any palindrome), the palindromic length of any finite substring is bounded. Frid et al. conjectured that all such infinite strings are (ultimately) periodic.

It is easy to compute \( PL(S) \) in quadratic time via dynamic programming. Alatabbi, Iliopoulos and Rahman \[1\] very recently gave a linear-time algorithm for computing a minimum factorization of \( S \) into maximal palindromic substrings, when such a factorization exists; it does not exist for, e.g., \( abaca \). Even when such a factorization exists, it may consist of more than \( PL(S) \) substrings; e.g., \( abbaabaabbba \) can be factored into \( abba, aba \) and \( abba \) but cannot be factored into fewer than four maximal palindromic substrings. In this paper, we give an \( O(n \log n) \)-time and \( O(n) \)-space algorithm for factoring \( S \) into \( PL(S) \) palindromic substrings.

### 2. A Simple Quadratic Algorithm

We start by describing a simple algorithm for computing \( PL(S) \) in \( O(n^2) \) time and \( O(n) \) space using the observation that, for \( 1 \leq j \leq n \),

\[
PL(S[1..j]) = \min_{i} \{ PL(S[1..i - 1]) + 1 \quad : \quad i \leq j, \ S[i..j] \text{ is a palindrome} \}.
\]

We compute and store an array \( PL[0..n] \), where \( PL[0] = 0 \) and \( PL[i] = PL(S[1..i]) \) for \( i \geq 1 \). At each step \( j \), we compute the set \( P_j \) of the starting positions of all palindromes ending at \( j \) from the set \( P_{j-1} \) using the observation that \( S[i..j] \), \( i + 1 \leq j - 1 \), is a palindrome if and only if \( S[i+1..j-1] \) is a palindrome and \( S[i] = S[j] \). The algorithm is given in Figure 1.

The space requirement is clearly \( O(n) \). During the \( j \)th step of the algorithm, we use time \( O(|P_j| + |P_{j-1}|) \), so for all the steps we use total time proportional to the number of palindromic substrings in \( S \). For most strings the time is linear, but the worst case is quadratic, e.g., for \( S = a^n \) or \( S = (ab)^{n/2} \).

It is straightforward to modify the algorithm so that it produces an actual minimum palindromic factorization of \( S \), without increasing the running time or space by more than a constant factor.
Algorithm Palindromic-length($S[1..n]$)
1: $PL[0] = 0$
2: $P \leftarrow \emptyset$
3: for $j \leftarrow 1$ to $n$ do
4:   $P' \leftarrow \emptyset$
5:   foreach $i \in P$ do
6:     if $i > 1$ and $S[i-1] = S[j]$ then
7:        $P' \leftarrow P' \cup \{i-1\}$
8:     if $j > 1$ and $S[j-1] = S[j]$ then
9:        $P' \leftarrow P' \cup \{j-1\}$
10:    $P \leftarrow P' \cup \{j\}$
11:    $PL[j] \leftarrow j$
12:   foreach $i \in P$ do
13:      $PL[j] \leftarrow \min(PL[j], PL[i-1] + 1)$
14: return $PL[n]$

Figure 1: A simple quadratic-time algorithm for computing the palindromic length.

3. Faster Computation of Palindromes

In this section, we describe a representation of the set $P_j$ of palindromes ending at $j$ that needs only $O(\log j)$ space, independent of the size of $P_j$, and can be computed in $O(\log j)$ time from the corresponding representation of $P_{j-1}$. The representation is based on combinatorial properties of palindromes.

A string $y$ is a border of a string $x$ if $y$ is both a prefix of $x$ and a suffix of $x$, and a proper border if $y \neq x$.

The following easy lemmas establish a connection between borders and palindromes.

Lemma 1. Let $y$ be a suffix of a palindrome $x$. Then $y$ is a border of $x$ iff $y$ is a palindrome.

Lemma 2. Let $x$ be a string with a border $y$ such that $|x| \leq 2|y|$. Then $x$ is a palindrome iff $y$ is a palindrome.

A positive integer $p \leq |x|$ is a period of a string $x$ if there exists a string $w$ of length $p$ such that $x$ is a factor of $w^\infty$. It is well known that $y$ is a proper border of $x$ if and only if $|x| - |y|$ is a period of $x$, which implies the following connection between periods and palindromes.

Lemma 3. Let $y$ be a proper suffix of a palindrome $x$. Then $|x| - |y|$ is a period of $x$ iff $y$ is a palindrome. In particular, $|x| - |y|$ is the smallest period of $x$ iff $y$ is the longest palindromic proper suffix of $x$.

Now we are ready to state and prove the key combinatorial property of palindromic suffixes.
Lemma 4. Let \( x \) be a palindrome, \( y \) the longest palindromic proper suffix of \( x \) and \( z \) the longest palindromic proper suffix of \( y \). Let \( u \) and \( v \) be strings such that \( x = uy \) and \( y = vz \).

Then

1. \(|u| \geq |v|\);
2. if \(|u| > |v|\) then \(|u| > |z|\);
3. if \(|u| = |v|\) then \( u = v \).

Proof. (1) By Lemma 3, \(|u| = |x| - |y|\) is the smallest period of \( x \), and \(|v| = |y| - |z|\) is the smallest period of \( y \). Since \( y \) is a factor of \( x \), either \(|u| > |y| > |v|\) or \(|u| \) is a period of \( y \) too, and thus it cannot be smaller than \(|v|\).

(2) By Lemma 1, \( y \) is a border of \( x \) and thus \( v \) is a prefix of \( x \). Let \( w \) be a string such that \( x = vw \). Then \( z \) is a border of \( w \) and \(|w| = |zu|\), see Figure 2. Since we assume \(|u| > |v|\), we must have \(|w| > |y|\). Suppose to the contrary that \(|u| \leq |z|\). Then \(|w| = |zu| \leq 2|z|\), and by Lemma 2 \( w \) is a palindrome. But this contradicts \( y \) being the longest palindromic proper suffix of \( x \).

(3) In the proof of (2) we saw that \( v \) is a prefix of \( x \), and so is \( u \) by definition. Thus \( u = v \) if \(|u| = |v|\).

![Figure 2: Proof of Lemma 4(2): if \(|u| > |v|\) and \(|u| \leq |z|\) then \( w \) is a palindromic proper suffix of \( x \) longer than \( y \).](image)

We will use the above lemma to establish the properties of the set \( P_j \). Let \( P_j = \{p_1, p_2, \ldots, p_m\} \) with \( p_1 < p_2 < \cdots < p_m \). By gap we mean the difference \( p_i - p_{i-1} \) of two consecutive values in \( P_j \).

Lemma 5. The sequence of gaps in \( P_j \) is non-increasing and there are at most \( O(\log j) \) distinct gaps.

Proof. For any \( i \in [2..m - 1] \), if we let \( x = S[p_{i-1}..j] \), \( y = S[p_i..j] \) and \( z = S[p_{i+1}..j] \), we have the situation of Lemma 4 with gaps of \(|u|\) and \(|v|\). The sequence of gaps is non-increasing by Lemma 3(1). If we have a change of gap,
i.e., \(|u| > |v|\), we must have \(|x| > |u| + |z| > 2|z|\) by Lemma 12, i.e., the length of the palindromic suffix is halved in two steps. This cannot happen more than \(\log j\) times.

We will partition the set \(P_j\) by the gaps into \(\mathcal{O}(\log j)\) consecutive subsets, each of which can be represented in constant space since it forms an arithmetic progression. For any positive integer \(\Delta\), we define \(P_{j,\Delta} = \{p_i : 1 < i \leq m, p_i - p_{i-1} = \Delta\}\), and \(P_{j,\infty} = \{p_1\}\). Each non-empty \(P_{j,\Delta}\) is represented by the triple \((\min P_{j,\Delta}, \Delta, |P_{j,\Delta}|)\). Let \(G_j\) be the list of such triples in decreasing order of \(\Delta\).

The list \(G_j\) is a full representation of \(P_j\) of size \(\mathcal{O}(\log j)\). We will show that \(G_j\) can be computed from \(G_{j-1}\) in \(\mathcal{O}(|G_{j-1}|)\) time. In the quadratic-time algorithm, each element \(i\) of \(P_{j-1}\) was either eliminated or replaced by \(i - 1\) in \(P_j\). The following lemma shows that the decision to eliminate or replace can be made simultaneously for all elements of a partition \(P_{j-1,\Delta}\).

**Lemma 6.** Let \(p_i\) and \(p_{i+1}\) be two consecutive elements of \(P_{j-1,\Delta}\). Then \(p_i - 1 \in P_j\) iff \(p_{i+1} - 1 \in P_j\).

**Proof.** By definition, \(p_{i+1} - p_i = \Delta\), and the predecessor of \(p_i\) in \(P_j\) is \(p_{i-1} = p_i - \Delta\). Using the definitions from the proof of Lemma 5, we have the situation of Lemma 3, which implies that \(S[p_i - 1] = S[p_{i+1} - 1] = c\). Thus, \(p_i - 1 \in P_j\) iff \(S[j] = c\) iff \(p_{i+1} - 1 \in P_j\). \(\square\)

Thus, when computing \(G_j\), each triple \((i, \Delta, k) \in G_{j-1}\) will be either eliminated or replaced by \((i - 1, \Delta, k)\). The resulting sequence of triples is

\[
G_j' = \{(i - 1, \Delta, k) : (i, \Delta, k) \in G_{j-1}, i > 1, \text{ and } S[i - 1] = S[j]\},
\]

which is a full representation of all palindromes longer than two in \(P_j\).

However, the triples in \(G_j'\) may no longer perfectly correspond to the partitions \(P_{j,\Delta}\), because the gaps may have changed. Specifically, if the smallest element \(p_i\) in \(P_{j-1,\Delta}\) is replaced by \(p_i - 1\) but its predecessor \(p_{i-1} = p_i - \Delta\) in \(P_{j-1}\) is eliminated, then \(p_i - 1\) is not in \(P_{j,\Delta}\) but it is, at this point, represented by the triple \((p_i - 1, \Delta, k)\). Note that only the smallest element of each partition can be affected by this. In such cases, we separate the first element into its own triple, i.e., we replace \((p_i - 1, \Delta, k)\) with \((p_i - 1, \Delta', 1)\) and if \(k > 1\) \((p_i - 1 + \Delta, \Delta, k - 1)\), where \(\Delta'\) is the new gap preceding \(p_i - 1\) in \(P_j\). We will also add separate triples to represent palindromes of lengths one and (possibly) two.

Let \(G_j''\) be the sequence of triples obtained from \(G_j'\) by the above process (see lines 8–21 in Figure 5). It represents exactly the palindromes in \(P_j\) and the \(\Delta\)-values are now correct, but there may be multiple triples with the same \(\Delta\). Thus we obtain the final sequence \(G_j\) by from \(G_j''\) by merging triples with the same \(\Delta\).

The full procedure for computing \(G_j\) from \(G_{j-1}\) is shown on lines 4–30 in Figure 5. Each triple is processed in constant time and the number of triples never exceeds \(\mathcal{O}(|G_{j-1}|)\).

**Lemma 7.** \(G_j\) can be computed from \(G_{j-1}\) in \(\mathcal{O}(|G_{j-1}|) = \mathcal{O}(\log j)\) time.
4. Faster Factorization

In this section, we will show how to compute $PL[j]$ from $PL[0..j−1]$ and $G_j$ in $O(|G_j|)$ time. The key to fast computation of $G_j$ was the close relation between $P_{j,\Delta}$ and $P_{j−1,\Delta}$. Now we will rely on the relation of $P_{j,\Delta}$ and $P_{j−\Delta,\Delta}$ captured by the following result.

**Lemma 8.** If $(i, \Delta, k) \in G_j$ for $k \geq 2$, then $(i, \Delta, k−1) \in G_{j−\Delta}$.

**Proof.** By definition, $(i, \Delta, k) \in G_j$ is equivalent to saying that $P_{j,\Delta} = \{i, i + \Delta, \ldots, i + (k−1)\Delta\}$, and we need to show that $P_{j−\Delta,\Delta} = \{i, i + \Delta, \ldots, i + (k−2)\Delta\}$. We will show first that $P_{j−\Delta,\Delta} \cap [i−\Delta + 1..j−\Delta] = \{i, i + \Delta, \ldots, i + (k−2)\Delta\}$ and then that $P_{j−\Delta,\Delta} \cap [1..i−\Delta] = \emptyset$.

Since $y = S[i..j]$ and $x = S[i−\Delta..j]$ are palindromes and $y$ is the longest proper border of $x$, $S[i−\Delta..j−\Delta] = y = S[i..j]$. Thus for all $\ell \in [i..j]$, $\ell \in P_j$ iff $\ell−\Delta \in P_{j−\Delta}$ (see Figure 3). In particular, the gaps in both cases are the same and for all $\ell \in [i+1..j]$, $\ell \in P_{j,\Delta}$ iff $\ell−\Delta \in P_{j−\Delta,\Delta}$. Thus $P_{j−\Delta,\Delta} \cap [i−\Delta + 1..j−\Delta] = \{i, i + \Delta, \ldots, i + (k−2)\Delta\}$.

We still need to show that $P_{j−\Delta,\Delta} \cap [1..i−\Delta] = \emptyset$, which is true if and only if $i−2\Delta \notin P_{j−\Delta}$. Suppose to the contrary that $S[i−2\Delta..i−\Delta−1]$ is a palindrome and let $w = S[i−2\Delta..i−\Delta−1]$. Then $S[j−2\Delta + 1..j−\Delta] = w^R$, the reverse of $w$. Since $z = S[i−\Delta..j−\Delta]$ and $S[i−\Delta..j]$ are palindromes too, we have that $S[i−\Delta..i−1] = w$ and $S[j−\Delta + 1..j] = w^R$. Finally, since $z$ is a palindrome, $S[i−2\Delta..j] = wzw^R$ is a palindrome (see Figure 3). This implies that $i−2\Delta \in P_j$ and thus $i−\Delta \in P_{j,\Delta}$ which is a contradiction. \qed

![Figure 3: Proof of Lemma 3](image-url)
By the above lemma, \( P_{j,\Delta} = P_{j-\Delta,\Delta} \cup \{ \max P_{j,\Delta} \} \) whenever \(|P_{j,\Delta}| \geq 2\). Thus we can compute \( PL_{j,\Delta} = \min\{ PL[i-1] + 1 : i \in P_{j,\Delta} \} \) from \( PL_{j-\Delta,\Delta} \) in constant time. We will store the value \( PL_{j,\Delta} \) in an array \( GPL[1..n] \) at the position \( m = \min P_{j,\Delta} - \Delta \). Note that \( m \) is the predecessor of \( \min P_{j,\Delta} \) in \( P_{j,\Delta} \) and the position is shared by \( PL_{j-\Delta,\Delta} \) (when \(|P_{j,\Delta}| \geq 2\)). The following lemma shows that the position is not overwritten by another value between the rounds \( j-\Delta \) and \( j \).

**Lemma 9.** Let \( m = \min P_{j,\Delta} - \Delta \). For all \( \ell \in [j-\Delta+1..j-1] \), \( m \notin P_{\ell} \).

*Proof.* Suppose to the contrary that \( m \in P_{\ell} \) for some \( \ell \in [j-\Delta+1..j-1] \), i.e., \( S[m..\ell] \) is a palindrome. Then \( S[m+h..\ell-h] \) for \( h = \ell - j + \Delta \) is a palindrome too (see Figure 4). Since \( \ell - h = j - \Delta \) and \( m < m + h < m + \Delta = \min P_{j-\Delta,\Delta} \), this contradicts \( m \) being the predecessor of \( \min P_{j-\Delta,\Delta} \) in \( P_{j-\Delta} \).

![Figure 4: Proof of Lemma 9](image)

The full algorithm is given in Figure 5. The running time of round \( j \) is \( O(|G_{j-1}| + |G_j|) \). Since \( |G_j| = O(\log j) \) for all \( j \), we obtain the following result.

**Theorem 10.** The palindromic length of a string of length \( n \) can be computed in \( O(n \log n) \) time and \( O(n) \) space.

The time \( O(n \log n) \) is in the worst case; for most strings the running time is linear.

As with the quadratic-time algorithm, the algorithm can be modified to produce an actual minimum palindromic factorization without an asymptotic increase in time or space complexities. We just need to store with each palindromic length in \( PL \) and \( GPL \) the length of the last palindrome in the corresponding minimum factorization.

**Acknowledgements** Many thanks to the organizers and participants of the Stringmasters 2013 workshops in Verona and Prague.
Algorithm Palindromic-length(S[1..n])
1: PL[0] = 0
2: G ← ()
3: for j ← 1 to n do
4:     G′ ← ()
5:     foreach (i, Δ, k) ∈ G do
6:         if i > 1 and S[i-1] = S[j] then
7:             G′.pushback((i-1, Δ, k))
8:     G″ ← ()
9:     r ← −j // makes i − r big enough to act as ∞
10:    foreach (i, Δ, k) ∈ G′ do
11:        if i − r ̸= Δ then
12:            G″.pushback((i, i − r, 1))
13:        if k > 1 then
14:            G″.pushback((i + Δ, Δ, k − 1))
15: else
16:     G″.pushback((i, Δ, k))
17:     r ← i + (k − 1)Δ
18:    if j > 1 and S[j-1] = S[j] then
19:        G″.pushback((j − 1, j − 1 − r, 1))
20:    r ← j − 1
21:    G″.pushback((j, j − r, 1))
22:    G ← ()
23: (i′, Δ′, k′) ← G″.popfront()
24: for (i, Δ, k) ∈ G″ do
25:    if Δ = Δ then
26:        k′ = k′ + k
27: else
28:     G.pushback((i′, Δ′, k′))
29:     (i′, Δ′, k′) ← (i, Δ, k)
30:     G.pushback((i′, Δ′, k′))
31: PL[j] ← j
32: for (i, Δ, k) ∈ G do
33:    r ← i + (k − 1)Δ
34:    m ← PL[r − 1] + 1
35:    if k > 1 then
36:        m ← min(m, GPL[i − ∆])
37:    if Δ ≤ i then
38:        GPL[i − ∆] ← m
39: PL[j] ← min(PL[j], m)
40: return PL[n]

Figure 5: Algorithm for computing the palindromic length in $O(n \log n)$ time.
References

[1] A. Alitabbi, C. S. Iliopoulos, and M. S. Rahman. Maximal palindromic factorization. In Proceedings of the Prague Stringology Conference (PSC), pages 70–77, 2013.

[2] J. P. Allouche, M. Baake, J. Cassaigne, and D. Damanik. Palindrome complexity. Theoretical Computer Science, 292(1):9–31, 2003.

[3] A. Apostolico, D. Breslauer, and Z. Galil. Parallel detection of all palindromes in a string. Theoretical Computer Science, 141(1–2):163–173, 1995.

[4] G. Fici and L. Q. Zamboni. On the least number of palindromes contained in an infinite word. Theoretical Computer Science, 481:1–8, 2013.

[5] A. E. Frid, S. Puzynina, and L. Zamboni. On palindromic factorization of words. Advances in Applied Mathematics, 50(5):737–748, 2013.

[6] D. Gusfield. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, 1997.

[7] J. Jeuring. The derivation of on-line algorithms, with an application to finding palindromes. Algorithmica, 11(2):146–184, 1994.

[8] J. Jeuring. Finding palindromes: Variants and algorithms. In P. Achten and P. Koopman, editors, The Beauty of Functional Code, pages 258–272. 2013.

[9] G. K. Manacher. A new linear-time “on-line” algorithm for finding the smallest initial palindrome of a string. Journal of the ACM, 22(3):346–351, 1975.