Finding Distinct Subpalindromes Online

Dmitry Kosolobov, Mikhail Rubinchik, and Arseny Shur

Ural Federal University
Ekaterinburg, Russia

Introduction

- A *palindrome* is a string that is equal to its reversal; for example, the string *aabcbaa* is a palindrome

Introduction

- A *palindrome* is a string that is equal to its reversal; for example, the string *aabcbaa* is a palindrome
- A *subpalindrome* of a string is a substring that is a palindrome

Introduction

- A *palindrome* is a string that is equal to its reversal; for example, the string *aabcbaa* is a palindrome
- A *subpalindrome* of a string is a substring that is a palindrome

- The well-known *Sturmian words* are characterized by their *palindromic complexity* (de Luca 1997, Droubay, Pirillo, 1999)
- There are several papers on the properties of *rich words*, see Glen, Justin, Widmer, Zambony (2009) (a word w with $|w|+1$ distinct subpalindromes is called *rich*).
- The class of rich words includes the *episturmian words* introduced by Droubay, Justin, Pirillo (2001).

Problem

Problem

Every string of length n contains at most $n+1$ distinct subpalindromes, including the empty string (Droubay, Justin, Pirillo, 2001).

Problem

Every string of length n contains at most $n+1$ distinct subpalindromes, including the empty string (Droubay, Justin, Pirillo, 2001).

Can one find all distinct subpalindromes of a string in linear time and space?

Problem

Every string of length n contains at most $n+1$ distinct subpalindromes, including the empty string (Droubay, Justin, Pirillo, 2001).

Can one find all distinct subpalindromes of a string in linear time and space?

This question was answered in the affirmative with an offline algorithm (Groult, Prieur, Richomme, 2010). In that paper, the existence of the corresponding online algorithm was stated as an open problem.

Theorem

There exists an online algorithm which finds all distinct subpalindromes in a string over a finite alphabet Σ in time

- $O(n|\Sigma|)$ *if* Σ *is unordered,*
- $O(n \log |\Sigma|)$ *if* Σ *is ordered,*

using linear space.
This algorithm is optimal in the comparison based computation model.

Lemma (Groult, Prieur, Richomme, 2010)

The set of all distinct subpalindromes of a string coincides with the set of longest palindromic suffixes of all prefixes of this string.

Example

The set of all subpalindromes of the string *aacba* is $\{a, b, c, aa\}$.

prefix	**longest palindromic suffix**
a	*a*
aa	*aa*
aac	*c*
aacb	*b*
aacba	*a*

Manacher's algorithm (1975)

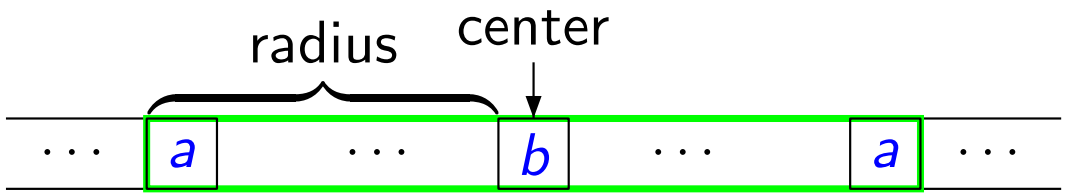
Manacher's algorithm (1975)

- A palindrome of even (resp. odd) length is referred to as an *even* (resp. *odd*) palindrome.

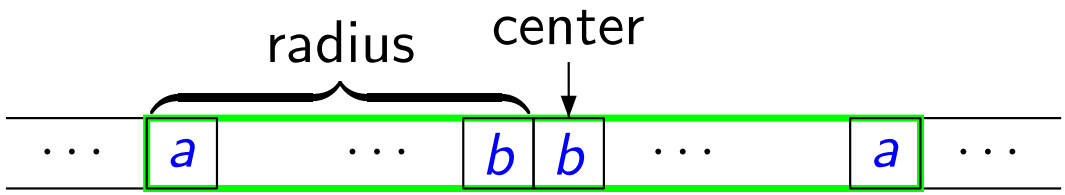
Manacher's algorithm (1975)

- A palindrome of even (resp. odd) length is referred to as an *even* (resp. *odd*) palindrome.
- The location of a subpalindrome in a string is determined by two numbers, *center* and *radius*:

odd subpalindromes:



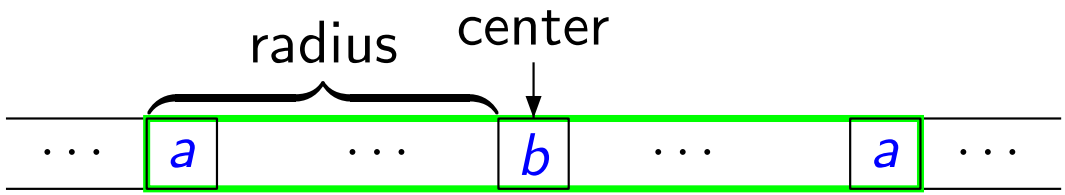
even subpalindromes:



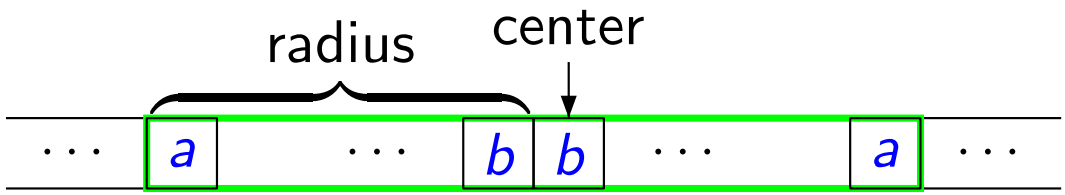
Manacher's algorithm (1975)

- A palindrome of even (resp. odd) length is referred to as an *even* (resp. *odd*) palindrome.
- The location of a subpalindrome in a string is determined by two numbers, *center* and *radius*:

odd subpalindromes:



even subpalindromes:



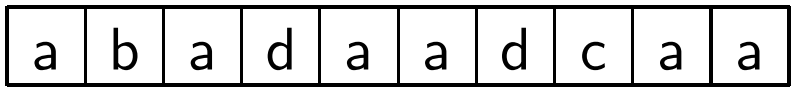
- Manacher's algorithm for odd (resp. even) palindromes computes online for the string $text[1..n]$ the array $Rad[1..n]$ such that $Rad[j]$ is the radius of the longest odd (resp. even) subpalindrome of $text[1..n]$ centered at j.

Manacher's algorithm pseudocode

The following modification of Manacher's algorithm computes online the longest odd palindromic suffix of a string. The algorithm for even case is analogous.

1: **procedure** AddLetter(c)
2: $s \leftarrow i - Rad[i]$ ▷ i is the center of $text[1..n]$ max odd suf-pal
3: $text[n+1] \leftarrow c$ ▷ append c to $text[1..n]$
4: **while** $i + Rad[i] \leqslant n$ **do**
5: $Rad[i] \leftarrow \min(Rad[s+n-i], n-i)$ ▷ $Rad[i]$ in $text[1..n]$
6: **if** $i + Rad[i] = n$ **and** $text[i-Rad[i]-1] = c$ **then**
7: $Rad[i] \leftarrow Rad[i] + 1$ ▷ extending the max suf-pal
8: **break** ▷ max odd suf-pal of $text[1..n+1]$ found
9: $i \leftarrow i + 1$ ▷ next candidate for the center
10: $n \leftarrow n + 1$ ▷ i is the center of max odd suf-pal of $text[1..n]$

Example

$w =$ 

Example

$w =$	a	b	a	d	a	a	d	c	a	a

Rad_{odd}										
Rad_{even}										
$LenPalSuf$										

Example

$w =$

a	b	a	d	a	a	d	c	a	a

Rad_{odd}	0	1	0	0						
Rad_{even}	0	0	0	0						
$LenPalSuf$	1	1	3	1						

Example

$w =$

a	b	a	d	a	a	d	c	a	a

Rad_{odd}	0	1	0	0						
Rad_{even}	0	0	0	0						
$LenPalSuf$	1	1	3	1						

Example

$w =$	a	b	a	d	a	a	d	c	a	a

Rad_{odd}	0	1	0	1						
Rad_{even}	0	0	0	0						
$LenPalSuf$	1	1	3	1						

Example

$w =$	a	b	a	d	a	a	d	c	a	a

Rad_{odd}	0	1	0	1						
Rad_{even}	0	0	0	0						
$LenPalSuf$	1	1	3	1						

Example

$w =$	a	b	a	d	a	a	d	c	a	a

Rad_{odd}	0	1	0	1						
Rad_{even}	0	0	0	0	0					
$LenPalSuf$	1	1	3	1						

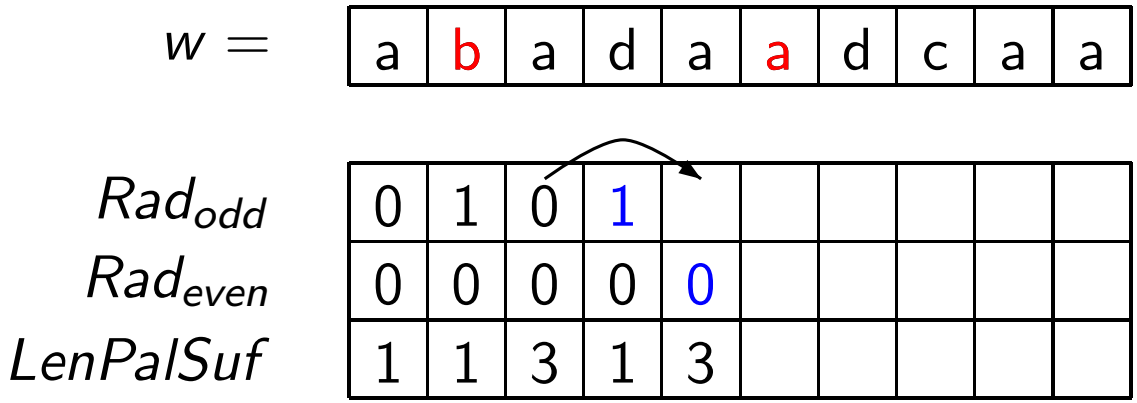
Example

$w =$

a	b	a	d	a	a	d	c	a	a

Rad_{odd}	0	1	0	1						
Rad_{even}	0	0	0	0	0					
$LenPalSuf$	1	1	3	1	3					

Example



Example

$w =$

a	b	a	d	a	a	d	c	a	a

Rad_{odd}	0	1	0	1	0					
Rad_{even}	0	0	0	0	0					
$LenPalSuf$	1	1	3	1	3					

Example

$w =$	a	b	a	d	a	a	d	c	a	a

Rad_{odd}	0	1	0	1	0	0				
Rad_{even}	0	0	0	0	0					
$LenPalSuf$	1	1	3	1	3					

Example

$w =$

a	b	a	d	a	a	d	c	a	a

Rad_{odd}	0	1	0	1	0	0				
Rad_{even}	0	0	0	0	1					
$LenPalSuf$	1	1	3	1	3					

Example

$w =$	a	b	a	d	a	a	d	c	a	a
Rad_{odd}	0	1	0	1	0	0				
Rad_{even}	0	0	0	0	1					
$LenPalSuf$	1	1	3	1	3	2				

Example

$w =$	a	b	a	d	a	a	d	c	a	a

Rad_{odd}	0	1	0	1	0	0	0			
Rad_{even}	0	0	0	0	1					
$LenPalSuf$	1	1	3	1	3	2				

Example

$w =$

a	b	a	d	a	a	d	c	a	a

Rad_{odd}	0	1	0	1	0	0	0			
Rad_{even}	0	0	0	0	2					
$LenPalSuf$	1	1	3	1	3	2				

Example

$w =$	a	b	a	d	a	a	d	c	a	a

Rad_{odd}	0	1	0	1	0	0	0			
Rad_{even}	0	0	0	0	2					
$LenPalSuf$	1	1	3	1	3	2	4			

Example

$w =$	a	b	a	d	a	a	d	c	a	a

Rad_{odd}	0	1	0	1	0	0	0	0	0	0
Rad_{even}	0	0	0	0	2	0	0	0	0	1
$LenPalSuf$	1	1	3	1	3	2	4	1	1	2

Example

$w =$	a	b	a	d	a	a	d	c	a	a

Rad_{odd}	0	1	0	1	0	0	0	0	0	0
Rad_{even}	0	0	0	0	2	0	0	0	0	1
$LenPalSuf$	1	1	3	1	3	2	4	1	1	2

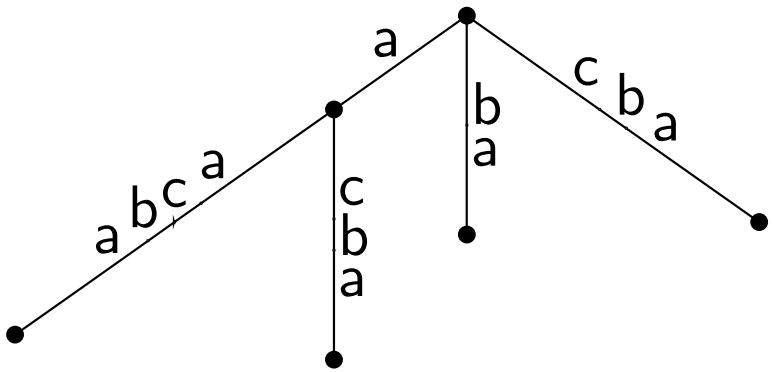
The last array gives us the multiset containing all subpalindromes of w:

$$\{a, b, aba, d, ada, aa, daad, c, a, aa\}.$$

How to remove all repetitions from this multiset?

Ukkonen's tree

Ukkonen's algorithm builds online the compressed suffix tree of a string. The algorithm requires $O(\log |\Sigma|)$ amortized time to append one letter to a string over alphabet Σ.

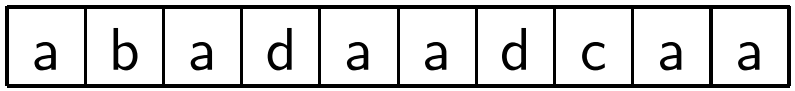


The compressed suffix tree for the word *aacba*

Ukkonen's algorithm efficiently implements the parameter *minUniqueSuff*: the length of the minimal suffix of the processed string such that this suffix occurs in this string only once (Ukkonen, 1995).

We don't interest in suffix tree itself we only need *minUniqueSuff* to verify whether the suffix of a string has another occurrence in this string.

Example (continued)

$w =$ 

Example (continued)

$w =$

a	b	a	d	a	a	d	c	a	a

LenPalSuf	1	1	3	1	3	2	4	1	1	2
MinUniqueSuff	1	1	2	1	2	2	3	1	2	3

Example (continued)

$w =$	a	b	a	d	a	a	d	c	a	a
LenPalSuf	1	1	3	1	3	2	4	1	1	2
MinUniqueSuff	1	1	2	1	2	2	3	1	2	3

The longest palindromic suffixes of the two longest prefixes of w do not contribute to the set of distinct subpalindromes.

Insert-only dictionary

A *dictionary* is a data structure D containing some set of elements from a given universe and designed for fast implementation of the basic operations like

- checking the membership of an element in the set;
- deleting an existing element;
- adding a new element.

Insert-only dictionary

A *dictionary* is a data structure D containing some set of elements from a given universe and designed for fast implementation of the basic operations like

- checking the membership of an element in the set;
- deleting an existing element;
- adding a new element.

An *insert-only dictionary* supports only the operation $\mathrm{insqry}(x)$, which checks membership and, if x is not in the dictionary, adds it.

Insert-only dictionary

A *dictionary* is a data structure D containing some set of elements from a given universe and designed for fast implementation of the basic operations like

- checking the membership of an element in the set;
- deleting an existing element;
- adding a new element.

An *insert-only dictionary* supports only the operation $\mathrm{insqry}(x)$, which checks membership and, if x is not in the dictionary, adds it.

Lemma

Suppose that the alphabet Σ consists of indivisible elements, $n \geq |\Sigma|$, and the insert-only dictionary D over Σ is initially empty. Then the sequence of n calls of insqry *requires, in the worst case, $\Omega(n \log |\Sigma|)$ time if Σ is ordered and $\Omega(n|\Sigma|)$ if Σ is unordered.*

- We reduce the problem of maintaining an insert-only dictionary to counting distinct palindromes in a string, thus proving the required lower bounds for the problem of finding (or even just counting) distinct subpalindromes.

- We reduce the problem of maintaining an insert-only dictionary to counting distinct palindromes in a string, thus proving the required lower bounds for the problem of finding (or even just counting) distinct subpalindromes.
- We assume that we have a black box algorithm that processes an input string letter by letter and outputs, after each step, the number of distinct palindromes in the string read so far.

- We reduce the problem of maintaining an insert-only dictionary to counting distinct palindromes in a string, thus proving the required lower bounds for the problem of finding (or even just counting) distinct subpalindromes.
- We assume that we have a black box algorithm that processes an input string letter by letter and outputs, after each step, the number of distinct palindromes in the string read so far.
- We can assume that the considered black box algorithm works in time $O(n \cdot f(m))$, where m is the size of the alphabet of the processed string and the function $f(m)$ is non-decreasing.

Lemma

Suppose that a, b are two different letters and $w = abx_1abx_2 \cdots abx_n$ is a string such that each x_i is a letter different from a and b. Then all nonempty subpalindromes of w are single letters.

Maintaining an insert-only dictionary

Let us describe how to process a sequence of n calls $\mathrm{insqry}(x_1), \ldots, \mathrm{insqry}(x_n)$ starting from the empty dictionary. Suppose that $x_i \notin \{a, b\}$ for every i.

Maintaining an insert-only dictionary

Let us describe how to process a sequence of n calls $\mathrm{insqry}(x_1), \ldots, \mathrm{insqry}(x_n)$ starting from the empty dictionary. Suppose that $x_i \notin \{a, b\}$ for every i.

- We feed the black box with a, b, and x_i (in this order).

Maintaining an insert-only dictionary

Let us describe how to process a sequence of n calls $\mathrm{insqry}(x_1), \ldots, \mathrm{insqry}(x_n)$ starting from the empty dictionary. Suppose that $x_i \notin \{a, b\}$ for every i.

- We feed the black box with a, b, and x_i (in this order).
- We get the output of the black box and check whether the number of distinct subpalindromes in its input string increased.

Maintaining an insert-only dictionary

Let us describe how to process a sequence of n calls $\mathrm{insqry}(x_1), \ldots, \mathrm{insqry}(x_n)$ starting from the empty dictionary. Suppose that $x_i \notin \{a, b\}$ for every i.

- We feed the black box with a, b, and x_i (in this order).
- We get the output of the black box and check whether the number of distinct subpalindromes in its input string increased.
- By Lemma, the increase happens if and only if x_i appears in the input string of the black box for the first time. Thus, we can immediately answer the query "$x_i \in D$?", and, moreover, x_i is now in the dictionary.

Maintaining an insert-only dictionary

Let us describe how to process a sequence of n calls $\mathrm{insqry}(x_1), \ldots, \mathrm{insqry}(x_n)$ starting from the empty dictionary. Suppose that $x_i \notin \{a, b\}$ for every i.

- We feed the black box with a, b, and x_i (in this order).
- We get the output of the black box and check whether the number of distinct subpalindromes in its input string increased.
- By Lemma, the increase happens if and only if x_i appears in the input string of the black box for the first time. Thus, we can immediately answer the query "$x_i \in D$?", and, moreover, x_i is now in the dictionary.

The overall time bound is $O(n \cdot f(m))$. We obtain $f(m) = \Omega(\log m)$ (resp., $f(m) = \Omega(m)$) in the case of ordered (resp., unordered) alphabet Σ.

Conclusion

Our approach shows that it is hardly possible to design a linear time and space online algorithm for the discussed problem even in stronger natural computation models such as the word-RAM model or cellprobe model. The reason is the resource restrictions of dictionaries. However, up to the moment we have proved no nontrivial lower bounds for the insert-only dictionary in more sophisticated models than the comparison based model.

Thank you for your attention!