Suffix arrays

Paweł Gawrychowski

University of Wrocław

The goal

A string is just a sequence of characters over an alphabet Σ, for example *bbababababaab*.

String indexing

We are given a very long string $w[1..n]$. We want to preprocess it in small space, so that later we can answer **MANY** queries of the following form: given a pattern $p[1..m]$, doest it occur in w? And if so, where?

Today we will see suffix arrays, which allow us to preprocess a string in $\mathcal{O}(n)$ space and time, so that later any query can be answered in time $\mathcal{O}(m + \log n)$, or (after some tweaks) even better.

This is also known as text indexing. One should keep in mind that by text we don't really mean a natural language text, as such texts are not very long. It's better to think about biological sequences, which are very long strings over $\Sigma = \{A, C, G, T\}$.

The goal

A string is just a sequence of characters over an alphabet Σ, for example *bbababababaab*.

String indexing

We are given a very long string $w[1..n]$. We want to preprocess it in small space, so that later we can answer **MANY** queries of the following form: given a pattern $p[1..m]$, doest it occur in w? And if so, where?

Today we will see suffix arrays, which allow us to preprocess a string in $\mathcal{O}(n)$ space and time, so that later any query can be answered in time $\mathcal{O}(m + \log n)$, or (after some tweaks) even better.

This is also known as text indexing. One should keep in mind that by text we don't really mean a natural language text, as such texts are not very long. It's better to think about biological sequences, which are very long strings over $\Sigma = \{A, C, G, T\}$.

The goal

A string is just a sequence of characters over an alphabet Σ, for example *bbababababaab*.

String indexing

We are given a very long string $w[1..n]$. We want to preprocess it in small space, so that later we can answer **MANY** queries of the following form: given a pattern $p[1..m]$, doest it occur in w? And if so, where?

Today we will see suffix arrays, which allow us to preprocess a string in $\mathcal{O}(n)$ space and time, so that later any query can be answered in time $\mathcal{O}(m + \log n)$, or (after some tweaks) even better.

This is also known as text indexing. One should keep in mind that by text we don't really mean a natural language text, as such texts are not very long. It's better to think about biological sequences, which are very long strings over $\Sigma = \{A, C, G, T\}$.

The goal

A string is just a sequence of characters over an alphabet Σ, for example *bbababababaab*.

String indexing

We are given a very long string $w[1..n]$. We want to preprocess it in small space, so that later we can answer **MANY** queries of the following form: given a pattern $p[1..m]$, doest it occur in w? And if so, where?

Today we will see suffix arrays, which allow us to preprocess a string in $\mathcal{O}(n)$ space and time, so that later any query can be answered in time $\mathcal{O}(m + \log n)$, or (after some tweaks) even better.

This is also known as text indexing. One should keep in mind that by text we don't really mean a natural language text, as such texts are not very long. It's better to think about biological sequences, which are very long strings over $\Sigma = \{A, C, G, T\}$.

Lexicographical comparison

$s \leq t$ if either s is a prefix of t, or s and t agree on the first $i-1$ positions, i.e., $s[1] = t[1]$, $s[2] = t[2]$, ..., $s[i-1] = t[i-1]$, and then $s[i] < t[i]$.

Observe that $\leq$ is a total order.

While assuming that the size of the alphabet is constant is not unusual here, in some applications we will work in a more general setting, where a string of length n consists of letters which are numbers from $\{1, \ldots, n\}$. (But not much larger!)

Lexicographical comparison

$s \leq t$ if either s is a prefix of t, or s and t agree on the first $i-1$ positions, i.e., $s[1] = t[1]$, $s[2] = t[2]$, ..., $s[i-1] = t[i-1]$, and then $s[i] < t[i]$.

Observe that $\leq$ is a total order.

While assuming that the size of the alphabet is constant is not unusual here, in some applications we will work in a more general setting, where a string of length n consists of letters which are numbers from $\{1, \ldots, n\}$. (But not much larger!)

Lexicographical comparison

$s \leq t$ if either s is a prefix of t, or s and t agree on the first $i-1$ positions, i.e., $s[1] = t[1]$, $s[2] = t[2]$, ..., $s[i-1] = t[i-1]$, and then $s[i] < t[i]$.

Observe that $\leq$ is a total order.

While assuming that the size of the alphabet is constant is not unusual here, in some applications we will work in a more general setting, where a string of length n consists of letters which are numbers from $\{1, \ldots, n\}$. (But not much larger!)

Now the suffix array is simply the lexicographically sorted list of all suffixes of a given word w.

$w = \texttt{mississippi}$

$SA[1] = 11 = \texttt{i}$
$SA[2] = 8 = \texttt{ippi}$
$SA[3] = 5 = \texttt{issippi}$
$SA[4] = 2 = \texttt{ississippi}$
$SA[5] = 1 = \texttt{mississippi}$
$SA[6] = 10 = \texttt{pi}$
$SA[7] = 9 = \texttt{ppi}$
$SA[8] = 7 = \texttt{sippi}$
$SA[9] = 4 = \texttt{sissippi}$
$SA[10] = 6 = \texttt{ssippi}$
$SA[11] = 3 = \texttt{ssissippi}$

Now the suffix array is simply the lexicographically sorted list of all suffixes of a given word w.

$w =$ mississippi

$SA[1] = 11 =$ i
$SA[2] = 8 =$ ippi
$SA[3] = 5 =$ issippi
$SA[4] = 2 =$ ississippi
$SA[5] = 1 =$ mississippi
$SA[6] = 10 =$ pi
$SA[7] = 9 =$ ppi
$SA[8] = 7 =$ sippi
$SA[9] = 4 =$ sissippi
$SA[10] = 6 =$ ssippi
$SA[11] = 3 =$ ssissippi

Why this is useful?

Generating all occurrences of a given pattern

Say that we want to output all occurrences of $p = \texttt{ippi}$. Can we say something about the structure of the set of all occurrences when we look at the sorted list of all suffixes?

Lemma

All occurrences of the same p create a contiguous fragment $SA[i], SA[i+1], \ldots, SA[j]$ of the suffix array.

The question is how to determine this fragment?

Why this is useful?

Generating all occurrences of a given pattern

Say that we want to output all occurrences of $p = \texttt{ippi}$. Can we say something about the structure of the set of all occurrences when we look at the sorted list of all suffixes?

Lemma

All occurrences of the same p create a contiguous fragment $SA[i], SA[i+1], \ldots, SA[j]$ of the suffix array.

The question is how to determine this fragment?

Why this is useful?

Generating all occurrences of a given pattern

Say that we want to output all occurrences of $p = \texttt{ippi}$. Can we say something about the structure of the set of all occurrences when we look at the sorted list of all suffixes?

Lemma

All occurrences of the same p create a contiguous fragment $SA[i], SA[i+1], \ldots, SA[j]$ of the suffix array.

The question is how to determine this fragment?

Why this is useful?

Generating all occurrences of a given pattern

Say that we want to output all occurrences of $p = \texttt{ippi}$. Can we say something about the structure of the set of all occurrences when we look at the sorted list of all suffixes?

Lemma

All occurrences of the same p create a contiguous fragment $SA[i], SA[i+1], \ldots, SA[j]$ of the suffix array.

The question is how to determine this fragment?

Lemma

$SA[i-1] < p \leq SA[i]$.

Recall that $SA[1] < SA[2] < \ldots < SA[n]$. Hence we can binary search for the value of i! And then verify if it corresponds to an occurrence, i.e., whether p is indeed a prefix of $SA[i]$.

How to determine j?

Lemma

$SA[i-1] < p \leq SA[i]$.

Recall that $SA[1] < SA[2] < \ldots < SA[n]$. Hence we can binary search for the value of i! And then verify if it corresponds to an occurrence, i.e., whether p is indeed a prefix of $SA[i]$.

How to determine j?

Lemma

$SA[i-1] < p \leq SA[i]$.

Recall that $SA[1] < SA[2] < \ldots < SA[n]$. Hence we can binary search for the value of i! And then verify if it corresponds to an occurrence, i.e., whether p is indeed a prefix of $SA[i]$.

How to determine j?

So far so good, but there are at least three questions:

- How much time the binary search takes?
- How much space do we need to store the suffix array?
- How much time do we need to compute the suffix array?

Storing the suffix array is easy: even though we think that $SA[i]$ is a word, it is really just a number denoting the starting position in w. Having the number we can access any letter of the word corresponding to $SA[i]$ in $\mathcal{O}(1)$ time. Hence the required space is $\mathcal{O}(n)$.

Binary searching is more tricky. There are just $\mathcal{O}(\log n)$ iterations, but each of them requires...

...$\mathcal{O}(m)$ time. Hence the total complexity is $\mathcal{O}(m \log n)$. Later we will see how to decrease it to $\mathcal{O}(m + \log n)$.

Finally, constructing the suffix array in a naive way would take $\mathcal{O}(n^2 \log n)$ time. Now we will see how to decrease it to $\mathcal{O}(n)$.

Storing the suffix array is easy: even though we think that $SA[i]$ is a word, it is really just a number denoting the starting position in w. Having the number we can access any letter of the word corresponding to $SA[i]$ in $\mathcal{O}(1)$ time. Hence the required space is $\mathcal{O}(n)$.
Binary searching is more tricky. There are just $\mathcal{O}(\log n)$ iterations, but each of them requires...
...$\mathcal{O}(m)$ time. Hence the total complexity is $\mathcal{O}(m \log n)$. Later we will see how to decrease it to $\mathcal{O}(m + \log n)$.
Finally, constructing the suffix array in a naive way would take $\mathcal{O}(n^2 \log n)$ time. Now we will see how to decrease it to $\mathcal{O}(n)$.

Storing the suffix array is easy: even though we think that $SA[i]$ is a word, it is really just a number denoting the starting position in w. Having the number we can access any letter of the word corresponding to $SA[i]$ in $\mathcal{O}(1)$ time. Hence the required space is $\mathcal{O}(n)$.
Binary searching is more tricky. There are just $\mathcal{O}(\log n)$ iterations, but each of them requires...
...$\mathcal{O}(m)$ time. Hence the total complexity is $\mathcal{O}(m \log n)$. Later we will see how to decrease it to $\mathcal{O}(m + \log n)$.
Finally, constructing the suffix array in a naive way would take $\mathcal{O}(n^2 \log n)$ time. Now we will see how to decrease it to $\mathcal{O}(n)$.

Storing the suffix array is easy: even though we think that $SA[i]$ is a word, it is really just a number denoting the starting position in w. Having the number we can access any letter of the word corresponding to $SA[i]$ in $\mathcal{O}(1)$ time. Hence the required space is $\mathcal{O}(n)$.
Binary searching is more tricky. There are just $\mathcal{O}(\log n)$ iterations, but each of them requires...
...$\mathcal{O}(m)$ time. Hence the total complexity is $\mathcal{O}(m \log n)$. Later we will see how to decrease it to $\mathcal{O}(m + \log n)$.
Finally, constructing the suffix array in a naive way would take $\mathcal{O}(n^2 \log n)$ time. Now we will see how to decrease it to $\mathcal{O}(n)$.

For the linear time suffix array construction algorithm we need two basic building blocks.

Radix sort

A sequence of n numbers from $\{1, 2, \ldots, k\}$ can be sorted in $\mathcal{O}(n + k)$ time. A sequence of n pairs from $\{1, \ldots, k\} \times \{1, \ldots, k\}$ can be sorted in the same complexity.

What about a sequence of triples?

Merging

Two sorted sequences $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_m$ can be merged in $\mathcal{O}(n + m)$ time.

For the linear time suffix array construction algorithm we need two basic building blocks.

Radix sort

A sequence of n numbers from $\{1, 2, \ldots, k\}$ can be sorted in $\mathcal{O}(n + k)$ time. A sequence of n pairs from $\{1, \ldots, k\} \times \{1, \ldots, k\}$ can be sorted in the same complexity.

What about a sequence of triples?

Merging

Two sorted sequences $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_m$ can be merged in $\mathcal{O}(n + m)$ time.

For the linear time suffix array construction algorithm we need two basic building blocks.

Radix sort

A sequence of n numbers from $\{1, 2, \ldots, k\}$ can be sorted in $\mathcal{O}(n + k)$ time. A sequence of n pairs from $\{1, \ldots, k\} \times \{1, \ldots, k\}$ can be sorted in the same complexity.

What about a sequence of triples?

Merging

Two sorted sequences $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_m$ can be merged in $\mathcal{O}(n + m)$ time.

For the linear time suffix array construction algorithm we need two basic building blocks.

Radix sort

A sequence of n numbers from $\{1, 2, \ldots, k\}$ can be sorted in $\mathcal{O}(n + k)$ time. A sequence of n pairs from $\{1, \ldots, k\} \times \{1, \ldots, k\}$ can be sorted in the same complexity.

What about a sequence of triples?

Merging

Two sorted sequences $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_m$ can be merged in $\mathcal{O}(n + m)$ time.

Let's start with an $\mathcal{O}(n^2)$ time algorithm.

$\mathcal{O}(n^2)$ algorithm

For each $i = n, n-1, n-2, \ldots, 1$ construct a sorted list L_i containing all $w[i..n]$, $w[i+1..n]$, $w[i+2..n]$, ... $w[n..n]$.

Assume that we have the list L_{i+1}, and want to construct the list for i. For each $j = i, i+1, \ldots, n$ construct a pair $(w[j], \mathrm{nr}_{i+1}[j+1])$, where $\mathrm{nr}_i(j)$ is the position of $w[j..n]$ on the list L_i. Then sort all pairs, and notice that their order determines L_{i+1}.

Let's start with an $\mathcal{O}(n^2)$ time algorithm.

$\mathcal{O}(n^2)$ algorithm

For each $i = n, n-1, n-2, \ldots, 1$ construct a sorted list L_i containing all $w[i..n]$, $w[i+1..n]$, $w[i+2..n]$, ... $w[n..n]$.

Assume that we have the list L_{i+1}, and want to construct the list for i. For each $j = i, i+1, \ldots, n$ construct a pair $(w[j], \mathrm{nr}_{i+1}[j+1])$, where $\mathrm{nr}_i(j)$ is the position of $w[j..n]$ on the list L_i. Then sort all pairs, and notice that their order determines L_{i+1}.

Let's start with an $\mathcal{O}(n^2)$ time algorithm.

$\mathcal{O}(n^2)$ algorithm

For each $i = n, n-1, n-2, \ldots, 1$ construct a sorted list L_i containing all $w[i..n]$, $w[i+1..n]$, $w[i+2..n]$, ... $w[n..n]$.

Assume that we have the list L_{i+1}, and want to construct the list for i. For each $j = i, i+1, \ldots, n$ construct a pair $(w[j], \mathrm{nr}_{i+1}[j+1])$, where $\mathrm{nr}_i(j)$ is the position of $w[j..n]$ on the list L_i. Then sort all pairs, and notice that their order determines L_{i+1}.

Now let's try to improve the complexity to $\mathcal{O}(n \log n)$.

When you think about the previous solution, some fragments of the word will be compared again... and again...

We apply the *doubling* method. To make the description easier, append n additional null characters to w, so that it is actually a word of length $2n$.

$\mathcal{O}(n \log n)$ algorithm

For each $i = 0, 1, 2, \ldots, \log n$ construct a sorted list L_i containing all $w[1..1 + 2^i - 1]$, $w[2..2 + 2^i - 1]$, ..., $w[n..n + 2^i - 1]$.

Assume that we have the list L_i, and want to construct the list for $i + 1$. For each $j = 1, 2, \ldots, n$ construct a pair $(\mathrm{nr}_i(j), \mathrm{nr}_i(j + 2^i))$, where $\mathrm{nr}_i(j)$ is the position of $w[j..j + 2^i - 1]$ on the list L_i. Then sort all pairs, and notice that their order determines L_{i+1}.

Now let's try to improve the complexity to $\mathcal{O}(n \log n)$.

When you think about the previous solution, some fragments of the word will be compared again... and again...

We apply the *doubling* method. To make the description easier, append n additional null characters to w, so that it is actually a word of length $2n$.

$\mathcal{O}(n \log n)$ algorithm

For each $i = 0, 1, 2, \ldots, \log n$ construct a sorted list L_i containing all $w[1..1 + 2^i - 1]$, $w[2..2 + 2^i - 1]$, ..., $w[n..n + 2^i - 1]$.

Assume that we have the list L_i, and want to construct the list for $i + 1$. For each $j = 1, 2, \ldots, n$ construct a pair $(\mathrm{nr}_i(j), \mathrm{nr}_i(j + 2^i))$, where $\mathrm{nr}_i(j)$ is the position of $w[j..j + 2^i - 1]$ on the list L_i. Then sort all pairs, and notice that their order determines L_{i+1}.

Now let's try to improve the complexity to $\mathcal{O}(n \log n)$.

When you think about the previous solution, some fragments of the word will be compared again... and again...

We apply the *doubling* method. To make the description easier, append n additional null characters to w, so that it is actually a word of length $2n$.

$\mathcal{O}(n \log n)$ algorithm

For each $i = 0, 1, 2, \ldots, \log n$ construct a sorted list L_i containing all $w[1..1 + 2^i - 1]$, $w[2..2 + 2^i - 1]$, ..., $w[n..n + 2^i - 1]$.

Assume that we have the list L_i, and want to construct the list for $i + 1$. For each $j = 1, 2, \ldots, n$ construct a pair $(\mathrm{nr}_i(j), \mathrm{nr}_i(j + 2^i))$, where $\mathrm{nr}_i(j)$ is the position of $w[j..j + 2^i - 1]$ on the list L_i. Then sort all pairs, and notice that their order determines L_{i+1}.

Now let's try to improve the complexity to $\mathcal{O}(n \log n)$.

When you think about the previous solution, some fragments of the word will be compared again... and again...

We apply the *doubling* method. To make the description easier, append n additional null characters to w, so that it is actually a word of length $2n$.

$\mathcal{O}(n \log n)$ algorithm

For each $i = 0, 1, 2, \ldots, \log n$ construct a sorted list L_i containing all $w[1..1 + 2^i - 1]$, $w[2..2 + 2^i - 1]$, ..., $w[n..n + 2^i - 1]$.

Assume that we have the list L_i, and want to construct the list for $i + 1$. For each $j = 1, 2, \ldots, n$ construct a pair $(\mathrm{nr}_i(j), \mathrm{nr}_i(j + 2^i))$, where $\mathrm{nr}_i(j)$ is the position of $w[j..j + 2^i - 1]$ on the list L_i. Then sort all pairs, and notice that their order determines L_{i+1}.

Now we are ready for the **real thing**.

Kärkkäinen and Sanders 2003

Suffix array can be constructed in $\mathcal{O}(n)$ time.

The idea is recursive. We will try to design the algorithm so that its running time can be expressed as $T(n) = T(\alpha n) + \mathcal{O}(n)$, where α is *some* constant less than 1.

Notice that the recursion solves to $T(n) = \mathcal{O}(n)$.

Now we are ready for the **real thing**.

Kärkkäinen and Sanders 2003

Suffix array can be constructed in $\mathcal{O}(n)$ time.

The idea is recursive. We will try to design the algorithm so that its running time can be expressed as $T(n) = T(\alpha n) + \mathcal{O}(n)$, where α is *some* constant less than 1.

Notice that the recursion solves to $T(n) = \mathcal{O}(n)$.

Now we are ready for the **real thing**.

Kärkkäinen and Sanders 2003

Suffix array can be constructed in $\mathcal{O}(n)$ time.

The idea is recursive. We will try to design the algorithm so that its running time can be expressed as $T(n) = T(\alpha n) + \mathcal{O}(n)$, where α is *some* constant less than 1.

Notice that the recursion solves to $T(n) = \mathcal{O}(n)$.

Now we are ready for the **real thing**.

Kärkkäinen and Sanders 2003

Suffix array can be constructed in $\mathcal{O}(n)$ time.

The idea is recursive. We will try to design the algorithm so that its running time can be expressed as $T(n) = T(\alpha n) + \mathcal{O}(n)$, where α is *some* constant less than 1.

Notice that the recursion solves to $T(n) = \mathcal{O}(n)$.

We partition all suffixes into three groups.

$$\begin{aligned} S_0 &= \{w[3..n], w[6..n], w[9..n], \ldots\} \\ S_1 &= \{w[1..n], w[4..n], w[7..n], \ldots\} \\ S_2 &= \{w[2..n], w[5..n], w[8..n], \ldots\} \end{aligned}$$

S_r are all suffixes that start at positions of the form $3k + r$.

The goal is to sort all suffixes. We could start with something simpler: sorting (separately) each S_r.

We partition all suffixes into three groups.

$$\begin{aligned} S_0 &= \{w[3..n], w[6..n], w[9..n], \ldots\} \\ S_1 &= \{w[1..n], w[4..n], w[7..n], \ldots\} \\ S_2 &= \{w[2..n], w[5..n], w[8..n], \ldots\} \end{aligned}$$

S_r are all suffixes that start at positions of the form $3k + r$.

The goal is to sort all suffixes. We could start with something simpler: sorting (separately) each S_r.

First trick

Say that we want to sort only S_1. We could split w into blocks of length 3, treat each block as a single letter, and recursively solve a smaller problem of size $\frac{n}{3}$.

$w[1]$	$w[2]$	$w[3]$	$w[4]$	$w[5]$	$w[6]$	$\cdots$	$w[n\text{-}2]$	$w[n\text{-}1]$	$w[n]$

We must be careful here: we promised that the input word of length $|w|$ will contain only letters $\{1, 2, \ldots, |w|\}$, and here we create triples of letters.

Alphabet renaming

Create a sorted list of all triples $(w[i], w[i+1], w[i+2])$ and then compute the number $\mathrm{nr}(i)$ of each triples there. This can be done in $\mathcal{O}(n)$ time using radix sort.

We append two null characters to w, so that the expression $(w[i], w[i+1], w[i+2])$ always makes sense.

$(w[1], w[2], w[3])$	$(w[4], w[5], w[6])$	$\cdots$	$(w[n\text{-}2], w[n\text{-}1], w[n])$

We must be careful here: we promised that the input word of length $|w|$ will contain only letters $\{1, 2, \ldots, |w|\}$, and here we create triples of letters.

Alphabet renaming

Create a sorted list of all triples $(w[i], w[i+1], w[i+2])$ and then compute the number $\mathrm{nr}(i)$ of each triples there. This can be done in $\mathcal{O}(n)$ time using radix sort.

We append two null characters to w, so that the expression $(w[i], w[i+1], w[i+2])$ always makes sense.

$(w[1], w[2], w[3])$	$(w[4], w[5], w[6])$	$\cdots$	$(w[n\text{-}2], w[n\text{-}1], w[n])$

We must be careful here: we promised that the input word of length $|w|$ will contain only letters $\{1, 2, \ldots, |w|\}$, and here we create triples of letters.

Alphabet renaming

Create a sorted list of all triples $(w[i], w[i+1], w[i+2])$ and then compute the number $\mathrm{nr}(i)$ of each triples there. This can be done in $\mathcal{O}(n)$ time using radix sort.

We append two null characters to w, so that the expression $(w[i], w[i+1], w[i+2])$ always makes sense.

$\mathrm{nr}(1)$	$\mathrm{nr}(4)$	$\cdots$	$\mathrm{nr}(n-2)$

We must be careful here: we promised that the input word of length $|w|$ will contain only letters $\{1, 2, \ldots, |w|\}$, and here we create triples of letters.

Alphabet renaming

Create a sorted list of all triples $(w[i], w[i+1], w[i+2])$ and then compute the number $\mathrm{nr}(i)$ of each triples there. This can be done in $\mathcal{O}(n)$ time using radix sort.

We append two null characters to w, so that the expression $(w[i], w[i+1], w[i+2])$ always makes sense.

OK, so we can sort S_1. Similarly, we can sort S_0 and S_2, but we cannot afford to sort all of them!

Second trick

Assuming that we have already sorted $S_1 \cup S_2$, we can sort $S_0 \cup S_1$ in $\mathcal{O}(n)$ time. For this we represent every suffix from $S_0 \cup S_1$ as a pair:

- $w[3k..n]$ becomes $(w[3k], w[3k+1..n])$,
- $w[3k+1..n]$ becomes $(w[3k+1], w[3k+2..n])$,

The order on pairs is clearly the same as the order on suffixes, so sorting the pairs allows us to sort the suffixes.

OK, so we can sort S_1. Similarly, we can sort S_0 and S_2, but we cannot afford to sort all of them!

Second trick

Assuming that we have already sorted $S_1 \cup S_2$, we can sort $S_0 \cup S_1$ in $\mathcal{O}(n)$ time. For this we represent every suffix from $S_0 \cup S_1$ as a pair:

- $w[3k..n]$ becomes $(w[3k], w[3k+1..n])$,
- $w[3k+1..n]$ becomes $(w[3k+1], w[3k+2..n])$,

The order on pairs is clearly the same as the order on suffixes, so sorting the pairs allows us to sort the suffixes.

Why this idea is all we need? We already have sorted $S_1 \cup S_2$. We can easily sort S_0 by replacing each suffix $w[3k..n]$ with the corresponding pair, and then sorting the pairs using radix sort. Just replace the second element of each pair with its position in the already known sorted sequence of all $S_1 \cup S_2$! Then we only have to merge two sorted sequences of length $\frac{1}{3}n$ and $\frac{2}{3}n$. This can be done in linear time, assuming that we can compare any two elements in $\mathcal{O}(1)$ time.

Why this idea is all we need? We already have sorted $S_1 \cup S_2$. We can easily sort S_0 by replacing each suffix $w[3k..n]$ with the corresponding pair, and then sorting the pairs using radix sort. Just replace the second element of each pair with its position in the already known sorted sequence of all $S_1 \cup S_2$! Then we only have to merge two sorted sequences of length $\frac{1}{3}n$ and $\frac{2}{3}n$. This can be done in linear time, assuming that we can compare any two elements in $\mathcal{O}(1)$ time.

Why this idea is all we need? We already have sorted $S_1 \cup S_2$. We can easily sort S_0 by replacing each suffix $w[3k..n]$ with the corresponding pair, and then sorting the pairs using radix sort. Just replace the second element of each pair with its position in the already known sorted sequence of all $S_1 \cup S_2$! Then we only have to merge two sorted sequences of length $\frac{1}{3}n$ and $\frac{2}{3}n$. This can be done in linear time, assuming that we can compare any two elements in $\mathcal{O}(1)$ time.

Why this idea is all we need? We already have sorted $S_1 \cup S_2$. We can easily sort S_0 by replacing each suffix $w[3k..n]$ with the corresponding pair, and then sorting the pairs using radix sort. Just replace the second element of each pair with its position in the already known sorted sequence of all $S_1 \cup S_2$! Then we only have to merge two sorted sequences of length $\frac{1}{3}n$ and $\frac{2}{3}n$. This can be done in linear time, assuming that we can compare any two elements in $\mathcal{O}(1)$ time.

So, how to compare any $w[3i..n] \in S_0$ with $w[j..n] \in S_1 \cup S_2$?

- $j = 3k + 1$, then represent $w[3i..n]$ as $(w[3i], w[3i + 1..n])$ and $w[j..n] = w[3k + 1..n]$ as $(w[3k + 1], w[3k + 2..n])$. $w[3i + 1..n]$ can be compared with $w[3k + 2]$ as they both belong to $S_1 \cup S_2$.
- $j = 3k + 2$, then represent $w[3i..n]$ as a triple $(w[3i], w[3i + 1], w[3i + 2..n])$ and $w[j..n] = w[3k + 2..n]$ as $(w[3k + 2], w[3(k + 1)], w[3(k + 1) + 1..n])$. $w[3i + 2..n]$ can be compared with $w[3(k + 1) + 1]$ as they both belong to $S_1 \cup S_2$.

Hence any two elements can be compared in $\mathcal{O}(1)$ time.

So, how to compare any $w[3i..n] \in S_0$ with $w[j..n] \in S_1 \cup S_2$?

- $j = 3k + 1$, then represent $w[3i..n]$ as $(w[3i], w[3i + 1..n])$ and $w[j..n] = w[3k + 1..n]$ as $(w[3k + 1], w[3k + 2..n])$. $w[3i + 1..n]$ can be compared with $w[3k + 2]$ as they both belong to $S_1 \cup S_2$.
- $j = 3k + 2$, then represent $w[3i..n]$ as a triple $(w[3i], w[3i + 1], w[3i + 2..n])$ and $w[j..n] = w[3k + 2..n]$ as $(w[3k + 2], w[3(k + 1)], w[3(k + 1) + 1..n])$. $w[3i + 2..n]$ can be compared with $w[3(k + 1) + 1]$ as they both belong to $S_1 \cup S_2$.

Hence any two elements can be compared in $\mathcal{O}(1)$ time.

So, how to compare any $w[3i..n] \in S_0$ with $w[j..n] \in S_1 \cup S_2$?

- $j = 3k + 1$, then represent $w[3i..n]$ as $(w[3i], w[3i + 1..n])$ and $w[j..n] = w[3k + 1..n]$ as $(w[3k + 1], w[3k + 2..n])$. $w[3i + 1..n]$ can be compared with $w[3k + 2]$ as they both belong to $S_1 \cup S_2$.
- $j = 3k + 2$, then represent $w[3i..n]$ as a triple $(w[3i], w[3i + 1], w[3i + 2..n])$ and $w[j..n] = w[3k + 2..n]$ as $(w[3k + 2], w[3(k + 1)], w[3(k + 1) + 1..n])$. $w[3i + 2..n]$ can be compared with $w[3(k + 1) + 1]$ as they both belong to $S_1 \cup S_2$.

Hence any two elements can be compared in $\mathcal{O}(1)$ time.

So, how to compare any $w[3i..n] \in S_0$ with $w[j..n] \in S_1 \cup S_2$?

- $j = 3k + 1$, then represent $w[3i..n]$ as $(w[3i], w[3i + 1..n])$ and $w[j..n] = w[3k + 1..n]$ as $(w[3k + 1], w[3k + 2..n])$. $w[3i + 1..n]$ can be compared with $w[3k + 2]$ as they both belong to $S_1 \cup S_2$.
- $j = 3k + 2$, then represent $w[3i..n]$ as a triple $(w[3i], w[3i + 1], w[3i + 2..n])$ and $w[j..n] = w[3k + 2..n]$ as $(w[3k + 2], w[3(k + 1)], w[3(k + 1) + 1..n])$. $w[3i + 2..n]$ can be compared with $w[3(k + 1) + 1]$ as they both belong to $S_1 \cup S_2$.

Hence any two elements can be compared in $\mathcal{O}(1)$ time.

We are almost done. The only remaining question is how to sort $S_1 \cup S_2$. We know how to sort S_1 and S_2 separately with a recursive call, but we need a stronger observation.

Third trick

The order on all $S_1 \cup S_2$ can be computed by sorting all suffixes of $w' = \mathrm{nr}(1)\mathrm{nr}(4)\mathrm{nr}(7)\ldots\mathrm{nr}(2)\mathrm{nr}(5)\mathrm{nr}(8)\ldots$.

Finally, we get an algorithm with the running time of the form $T(n) = T(\frac{2}{3}n) + \mathcal{O}(n)$, which is $\mathcal{O}(n)$. Nice! ☺

We are almost done. The only remaining question is how to sort $S_1 \cup S_2$. We know how to sort S_1 and S_2 separately with a recursive call, but we need a stronger observation.

Third trick

The order on all $S_1 \cup S_2$ can be computed by sorting all suffixes of $w' = \mathrm{nr}(1)\mathrm{nr}(4)\mathrm{nr}(7)\ldots\mathrm{nr}(2)\mathrm{nr}(5)\mathrm{nr}(8)\ldots$.

Finally, we get an algorithm with the running time of the form $T(n) = T(\frac{2}{3}n) + \mathcal{O}(n)$, which is $\mathcal{O}(n)$. Nice! ☺

We are almost done. The only remaining question is how to sort $S_1 \cup S_2$. We know how to sort S_1 and S_2 separately with a recursive call, but we need a stronger observation.

Third trick

The order on all $S_1 \cup S_2$ can be computed by sorting all suffixes of $w' = \mathrm{nr}(1)\mathrm{nr}(4)\mathrm{nr}(7)\ldots\mathrm{nr}(2)\mathrm{nr}(5)\mathrm{nr}(8)\ldots$.

Finally, we get an algorithm with the running time of the form $T(n) = T(\frac{2}{3}n) + \mathcal{O}(n)$, which is $\mathcal{O}(n)$. Nice! ☺

We are almost done. The only remaining question is how to sort $S_1 \cup S_2$. We know how to sort S_1 and S_2 separately with a recursive call, but we need a stronger observation.

Third trick

The order on all $S_1 \cup S_2$ can be computed by sorting all suffixes of $w' = \mathrm{nr}(1)\mathrm{nr}(4)\mathrm{nr}(7)\ldots\mathrm{nr}(2)\mathrm{nr}(5)\mathrm{nr}(8)\ldots$.

Finally, we get an algorithm with the running time of the form $T(n) = T(\frac{2}{3}n) + \mathcal{O}(n)$, which is $\mathcal{O}(n)$. Nice! ☺

Recall that our motivation for constructing the suffix array was that using it we can locate an occurrence of any pattern of length m in $\mathcal{O}(m \log n)$ time. Now we are almost ready to speed this up! The suffix array SA alone is not that useful. Usually it is augmented with the inverse suffix array SA^{-1}, where $SA^{-1}[i]$ is the position of $w[i..n]$ in SA, i.e., $SA[SA^{-1}[i]] = i$, and with a longest common prefix structure.

LCP

$\mathrm{lcp}(i, j)$ is the longest common prefix of $w[i..n]$ and $w[j..n]$. $\mathrm{lcp}[i]$ is the longest common prefix of the $(i-1)$-th and i-th suffix in the suffix array, or in other words $\mathrm{lcp}(SA[i-1], SA[i])$, with $\mathrm{lcp}[1]$ not defined.

Recall that our motivation for constructing the suffix array was that using it we can locate an occurrence of any pattern of length m in $\mathcal{O}(m \log n)$ time. Now we are almost ready to speed this up! The suffix array SA alone is not that useful. Usually it is augmented with the inverse suffix array SA^{-1}, where $SA^{-1}[i]$ is the position of $w[i..n]$ in SA, i.e., $SA[SA^{-1}[i]] = i$, and with a longest common prefix structure.

LCP

$\mathrm{lcp}(i, j)$ is the longest common prefix of $w[i..n]$ and $w[j..n]$. $\mathrm{lcp}[i]$ is the longest common prefix of the $(i-1)$-th and i-th suffix in the suffix array, or in other words $\mathrm{lcp}(SA[i-1], SA[i])$, with $\mathrm{lcp}[1]$ not defined.

Recall that our motivation for constructing the suffix array was that using it we can locate an occurrence of any pattern of length m in $\mathcal{O}(m \log n)$ time. Now we are almost ready to speed this up! The suffix array SA alone is not that useful. Usually it is augmented with the inverse suffix array SA^{-1}, where $SA^{-1}[i]$ is the position of $w[i..n]$ in SA, i.e., $SA[SA^{-1}[i]] = i$, and with a longest common prefix structure.

LCP

$\mathrm{lcp}(i, j)$ is the longest common prefix of $w[i..n]$ and $w[j..n]$. $\mathrm{lcp}[i]$ is the longest common prefix of the $(i-1)$-th and i-th suffix in the suffix array, or in other words $\mathrm{lcp}(SA[i-1], SA[i])$, with $\mathrm{lcp}[1]$ not defined.

$w =$ mississippi

$SA[1] =$	11	$=$	i
$SA[2] =$	8	$=$	ippi
$SA[3] =$	5	$=$	issippi
$SA[4] =$	2	$=$	ississippi
$SA[5] =$	1	$=$	mississippi
$SA[6] =$	10	$=$	pi
$SA[7] =$	9	$=$	ppi
$SA[8] =$	7	$=$	sippi
$SA[9] =$	4	$=$	sissippi
$SA[10] =$	6	$=$	ssippi
$SA[11] =$	3	$=$	ssissippi

What is $\mathrm{lcp}(8, 2)$?

$w =$ `mississippi`

$SA[1] =$	11	$=$	`i`			
$SA[2] =$	8	$=$	`ippi`	$\mathrm{lcp}[2]$	$=$	1
$SA[3] =$	5	$=$	`issippi`	$\mathrm{lcp}[3]$	$=$	1
$SA[4] =$	2	$=$	`ississippi`	$\mathrm{lcp}[4]$	$=$	4
$SA[5] =$	1	$=$	`mississippi`	$\mathrm{lcp}[5]$	$=$	0
$SA[6] =$	10	$=$	`pi`	$\mathrm{lcp}[6]$	$=$	0
$SA[7] =$	9	$=$	`ppi`	$\mathrm{lcp}[7]$	$=$	1
$SA[8] =$	7	$=$	`sippi`	$\mathrm{lcp}[8]$	$=$	0
$SA[9] =$	4	$=$	`sissippi`	$\mathrm{lcp}[9]$	$=$	2
$SA[10] =$	6	$=$	`ssippi`	$\mathrm{lcp}[10]$	$=$	1
$SA[11] =$	3	$=$	`ssissippi`	$\mathrm{lcp}[11]$	$=$	3

What is $\mathrm{lcp}(8, 2)$?

Lemma

$\mathrm{lcp}(i, j)$ is the minimum of all $\mathrm{lcp}[k]$ over $k = SA^{-1}[i] + 1$, $SA^{-1}[i] + 2$, ..., $SA^{-1}[j]$, assuming i is before j in the suffix array.

So what?

So, assuming that we know $\mathrm{lcp}[i]$, computing any $\mathrm{lcp}(i, j)$ requires just one range minimum query.

RMQ

Given an array $A[1..n]$, preprocess it so that the minimum of any fragment $A[i], A[i + 1], \ldots, A[j]$ can be computed efficiently.

Later we will see a linear time/space preprocessing allowing answering any query in $\mathcal{O}(1)$ time. For the time being assume that we know how to do that.

Lemma

$\mathrm{lcp}(i, j)$ is the minimum of all $\mathrm{lcp}[k]$ over $k = SA^{-1}[i] + 1$, $SA^{-1}[i] + 2$, ..., $SA^{-1}[j]$, assuming i is before j in the suffix array.

So what?

So, assuming that we know $\mathrm{lcp}[i]$, computing any $\mathrm{lcp}(i, j)$ requires just one range minimum query.

RMQ

Given an array $A[1..n]$, preprocess it so that the minimum of any fragment $A[i], A[i + 1], \ldots, A[j]$ can be computed efficiently.

Later we will see a linear time/space preprocessing allowing answering any query in $\mathcal{O}(1)$ time. For the time being assume that we know how to do that.

Lemma

$\mathrm{lcp}(i, j)$ is the minimum of all $\mathrm{lcp}[k]$ over $k = SA^{-1}[i] + 1$, $SA^{-1}[i] + 2$, ..., $SA^{-1}[j]$, assuming i is before j in the suffix array.

So what?

So, assuming that we know $\mathrm{lcp}[i]$, computing any $\mathrm{lcp}(i, j)$ requires just one range minimum query.

RMQ

Given an array $A[1..n]$, preprocess it so that the minimum of any fragment $A[i], A[i + 1], \ldots, A[j]$ can be computed efficiently.

Later we will see a linear time/space preprocessing allowing answering any query in $\mathcal{O}(1)$ time. For the time being assume that we know how to do that.

Lemma

$\mathrm{lcp}(i, j)$ is the minimum of all $\mathrm{lcp}[k]$ over $k = SA^{-1}[i] + 1$, $SA^{-1}[i] + 2$, ..., $SA^{-1}[j]$, assuming i is before j in the suffix array.

So what?

So, assuming that we know $\mathrm{lcp}[i]$, computing any $\mathrm{lcp}(i, j)$ requires just one range minimum query.

RMQ

Given an array $A[1..n]$, preprocess it so that the minimum of any fragment $A[i], A[i + 1], \ldots, A[j]$ can be computed efficiently.

Later we will see a linear time/space preprocessing allowing answering any query in $\mathcal{O}(1)$ time. For the time being assume that we know how to do that.

OK, but how to compute the array $\mathrm{lcp}[2], \mathrm{lcp}[3], \ldots, \mathrm{lcp}[n]$?

A trivial solution is to compute each of them separately, resulting in $\mathcal{O}(n^2)$ total complexity. It turns out that a small modification to this naive method improves the time to linear.

Lemma

For any suffix $w[i..n]$, $\mathrm{lcp}[SA^{-1}[i]] - 1 \leq \mathrm{lcp}[SA^{-1}[i+1]]$, assuming $i < n$ and neither $SA^{-1}[i]$ nor $SA^{-1}[i+1]$ is the first element of the suffix array.

Kasai et al. 2001

All lcp can be computed in (amortized) $\mathcal{O}(1)$ time per entry.

We compute $\mathrm{lcp}[SA^{-1}[i]]$ for $i = 1, 2, 3, \ldots$. At every step we naively compute the lcp between $w[i..n]$ and its predecessor in the suffix array, but we start the computation from the $\mathrm{lcp}[SA^{-1}[i-1]] - 1$-th letter!

OK, but how to compute the array $\mathrm{lcp}[2], \mathrm{lcp}[3], \ldots, \mathrm{lcp}[n]$?
A trivial solution is to compute each of them separately, resulting in $\mathcal{O}(n^2)$ total complexity. It turns out that a small modification to this naive method improves the time to linear.

Lemma

For any suffix $w[i..n]$, $\mathrm{lcp}[SA^{-1}[i]] - 1 \leq \mathrm{lcp}[SA^{-1}[i+1]]$, assuming $i < n$ and neither $SA^{-1}[i]$ nor $SA^{-1}[i+1]$ is the first element of the suffix array.

Kasai et al. 2001

All lcp can be computed in (amortized) $\mathcal{O}(1)$ time per entry.

We compute $\mathrm{lcp}[SA^{-1}[i]]$ for $i = 1, 2, 3, \ldots$. At every step we naively compute the lcp between $w[i..n]$ and its predecessor in the suffix array, but we start the computation from the $\mathrm{lcp}[SA^{-1}[i-1]] - 1$-th letter!

OK, but how to compute the array $\mathrm{lcp}[2], \mathrm{lcp}[3], \ldots, \mathrm{lcp}[n]$?
A trivial solution is to compute each of them separately, resulting in $\mathcal{O}(n^2)$ total complexity. It turns out that a small modification to this naive method improves the time to linear.

Lemma

For any suffix $w[i..n]$, $\mathrm{lcp}[SA^{-1}[i]] - 1 \leq \mathrm{lcp}[SA^{-1}[i+1]]$, assuming $i < n$ and neither $SA^{-1}[i]$ nor $SA^{-1}[i+1]$ is the first element of the suffix array.

Kasai et al. 2001

All lcp can be computed in (amortized) $\mathcal{O}(1)$ time per entry.

We compute $\mathrm{lcp}[SA^{-1}[i]]$ for $i = 1, 2, 3, \ldots$. At every step we naively compute the lcp between $w[i..n]$ and its predecessor in the suffix array, but we start the computation from the $\mathrm{lcp}[SA^{-1}[i-1]] - 1$-th letter!

OK, but how to compute the array $\mathrm{lcp}[2], \mathrm{lcp}[3], \ldots, \mathrm{lcp}[n]$?
A trivial solution is to compute each of them separately, resulting in $\mathcal{O}(n^2)$ total complexity. It turns out that a small modification to this naive method improves the time to linear.

Lemma

For any suffix $w[i..n]$, $\mathrm{lcp}[SA^{-1}[i]] - 1 \leq \mathrm{lcp}[SA^{-1}[i+1]]$, assuming $i < n$ and neither $SA^{-1}[i]$ nor $SA^{-1}[i+1]$ is the first element of the suffix array.

Kasai et al. 2001

All lcp can be computed in (amortized) $\mathcal{O}(1)$ time per entry.

We compute $\mathrm{lcp}[SA^{-1}[i]]$ for $i = 1, 2, 3, \ldots$. At every step we naively compute the lcp between $w[i..n]$ and its predecessor in the suffix array, but we start the computation from the $\mathrm{lcp}[SA^{-1}[i-1]] - 1$-th letter!

And recall that we wanted to use the suffix array to locate any (or all) occurrences of a given pattern.

Searching for an occurrence of p

We want to locate the smallest i such that $SA[i] \geq p$. Then either $SA[i]$ begins with p, and hence p occurs at position i, or there is no occurrence at all.

Binary search

Binary search uses $\log n$ iterations, but each of them might cost even $\Omega(m)$ operations! Hence the whole procedure is $\mathcal{O}(m \log n)$.

And recall that we wanted to use the suffix array to locate any (or all) occurrences of a given pattern.

Searching for an occurrence of p

We want to locate the smallest i such that $SA[i] \geq p$. Then either $SA[i]$ begins with p, and hence p occurs at position i, or there is no occurrence at all.

Binary search

Binary search uses $\log n$ iterations, but each of them might cost even $\Omega(m)$ operations! Hence the whole procedure is $\mathcal{O}(m \log n)$.

And recall that we wanted to use the suffix array to locate any (or all) occurrences of a given pattern.

Searching for an occurrence of p

We want to locate the smallest i such that $SA[i] \geq p$. Then either $SA[i]$ begins with p, and hence p occurs at position i, or there is no occurrence at all.

Binary search

Binary search uses $\log n$ iterations, but each of them might cost even $\Omega(m)$ operations! Hence the whole procedure is $\mathcal{O}(m \log n)$.

Now the question is whether we can do better. It seems that we are wasting lots of time comparing very similar blocks of texts again and again. Not cool!

lcp again

Recall that $\mathrm{lcp}(i, j)$ is the longest common prefix of the suffixes $w[i..n]$ and $w[j..n]$. We know how to compute all $\mathrm{lcp}[i]$, and we observed that computing $\mathrm{lcp}(i, j)$ reduces to the so-called Range Minimum Query on the $\mathrm{lcp}[i]$ array.

For the time being assume that we know how to answer RMQ queries on any array in $\mathcal{O}(1)$ time after linear preprocessing. Then we can compute any $\mathrm{lcp}(i, j)$ in $\mathcal{O}(1)$ time. Can this help us to speed up the binary searching?

Now the question is whether we can do better. It seems that we are wasting lots of time comparing very similar blocks of texts again and again. Not cool!

lcp again

Recall that $\mathrm{lcp}(i,j)$ is the longest common prefix of the suffixes $w[i..n]$ and $w[j..n]$. We know how to compute all $\mathrm{lcp}[i]$, and we observed that computing $\mathrm{lcp}(i,j)$ reduces to the so-called Range Minimum Query on the $\mathrm{lcp}[i]$ array.

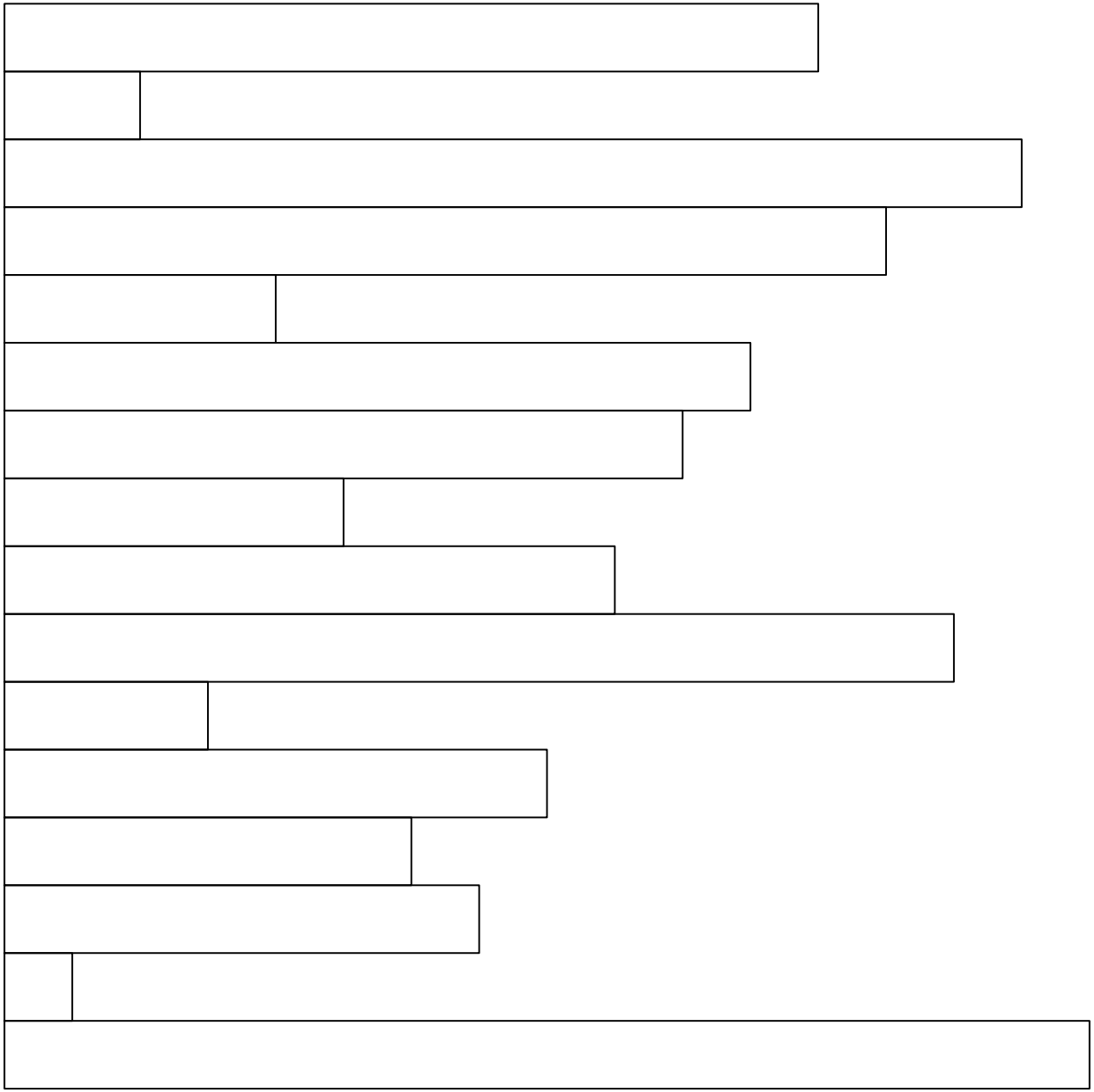
For the time being assume that we know how to answer RMQ queries on any array in $\mathcal{O}(1)$ time after linear preprocessing. Then we can compute any $\mathrm{lcp}(i,j)$ in $\mathcal{O}(1)$ time. Can this help us to speed up the binary searching?

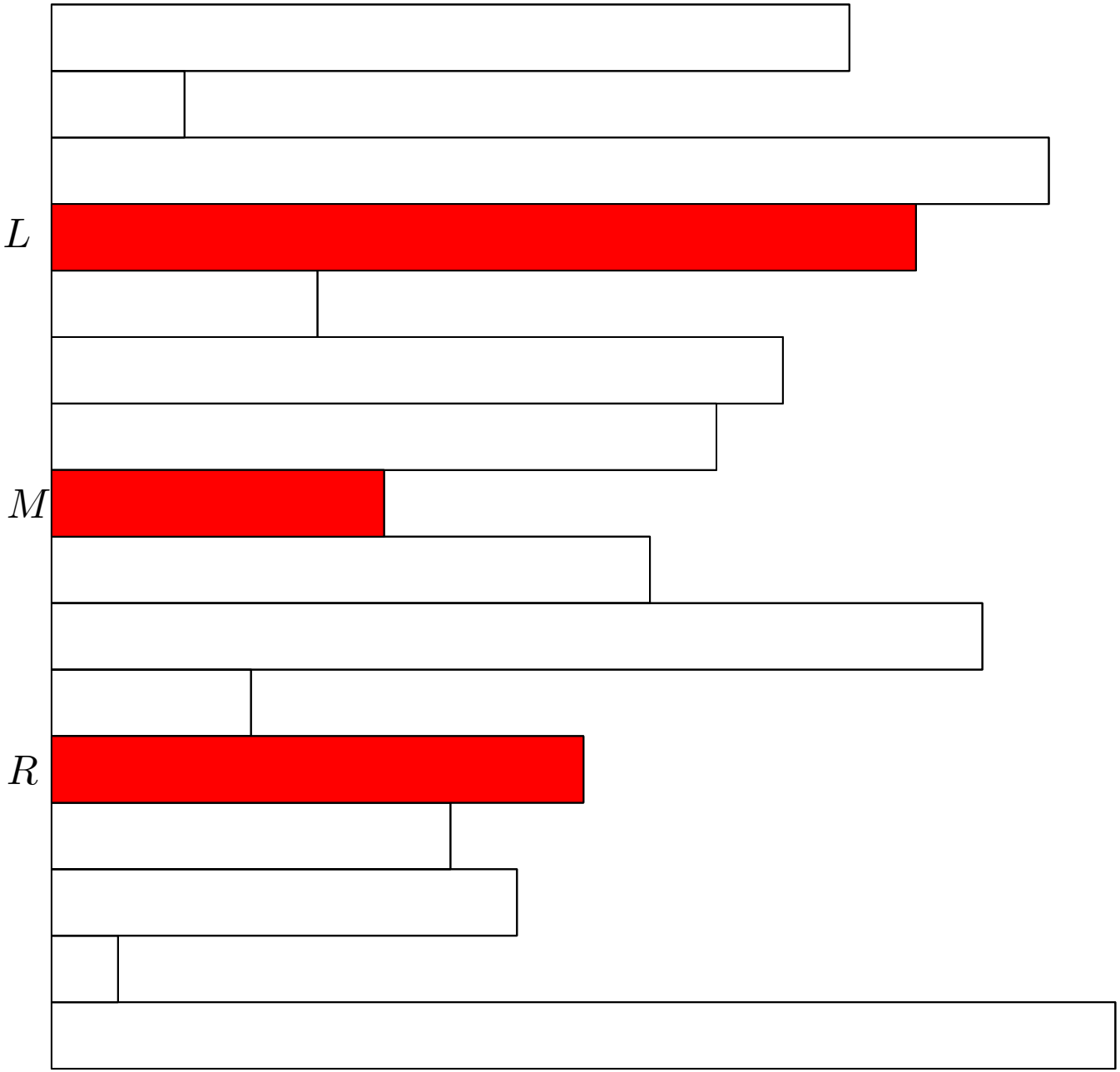
Now the question is whether we can do better. It seems that we are wasting lots of time comparing very similar blocks of texts again and again. Not cool!

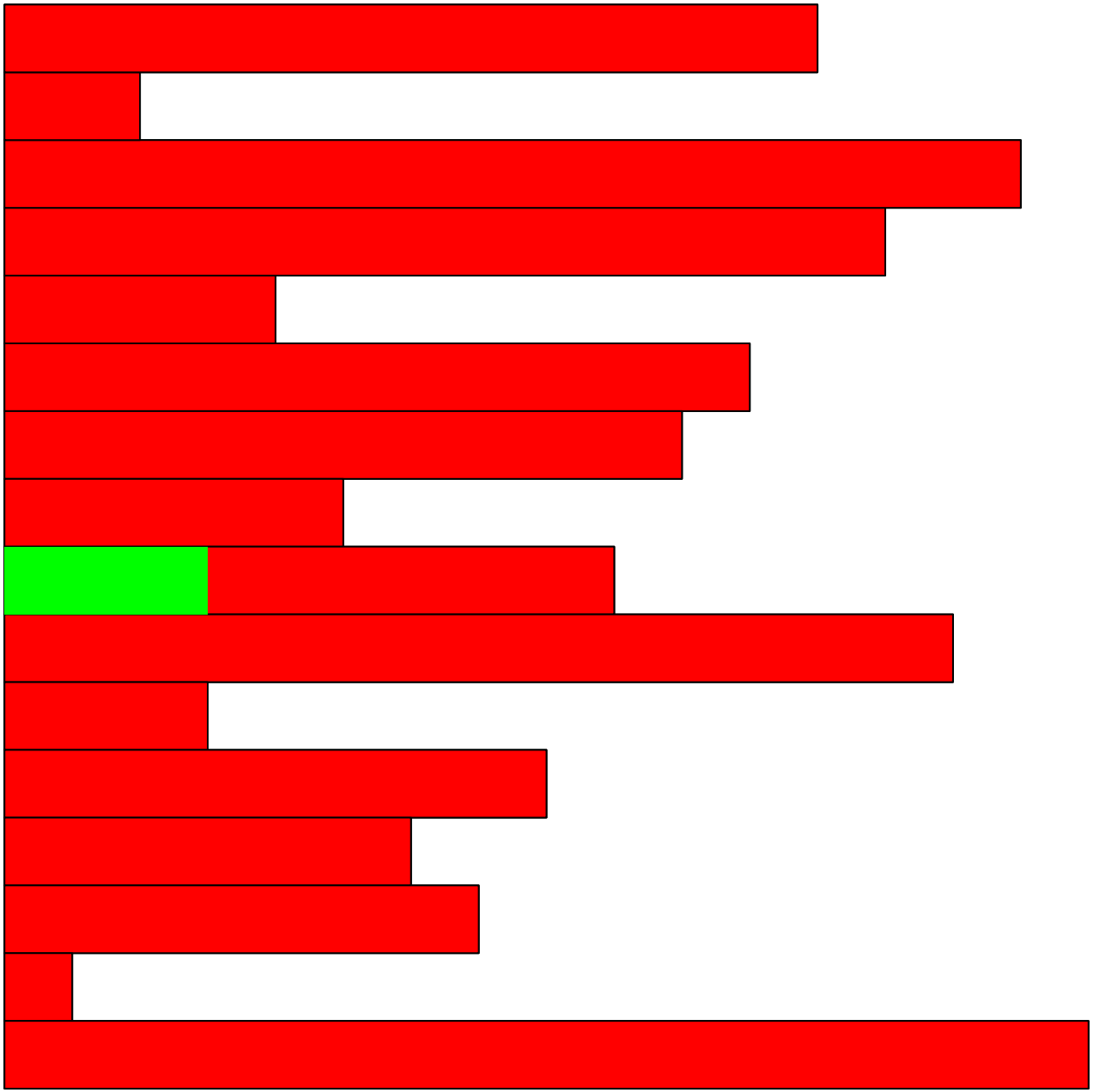
lcp again

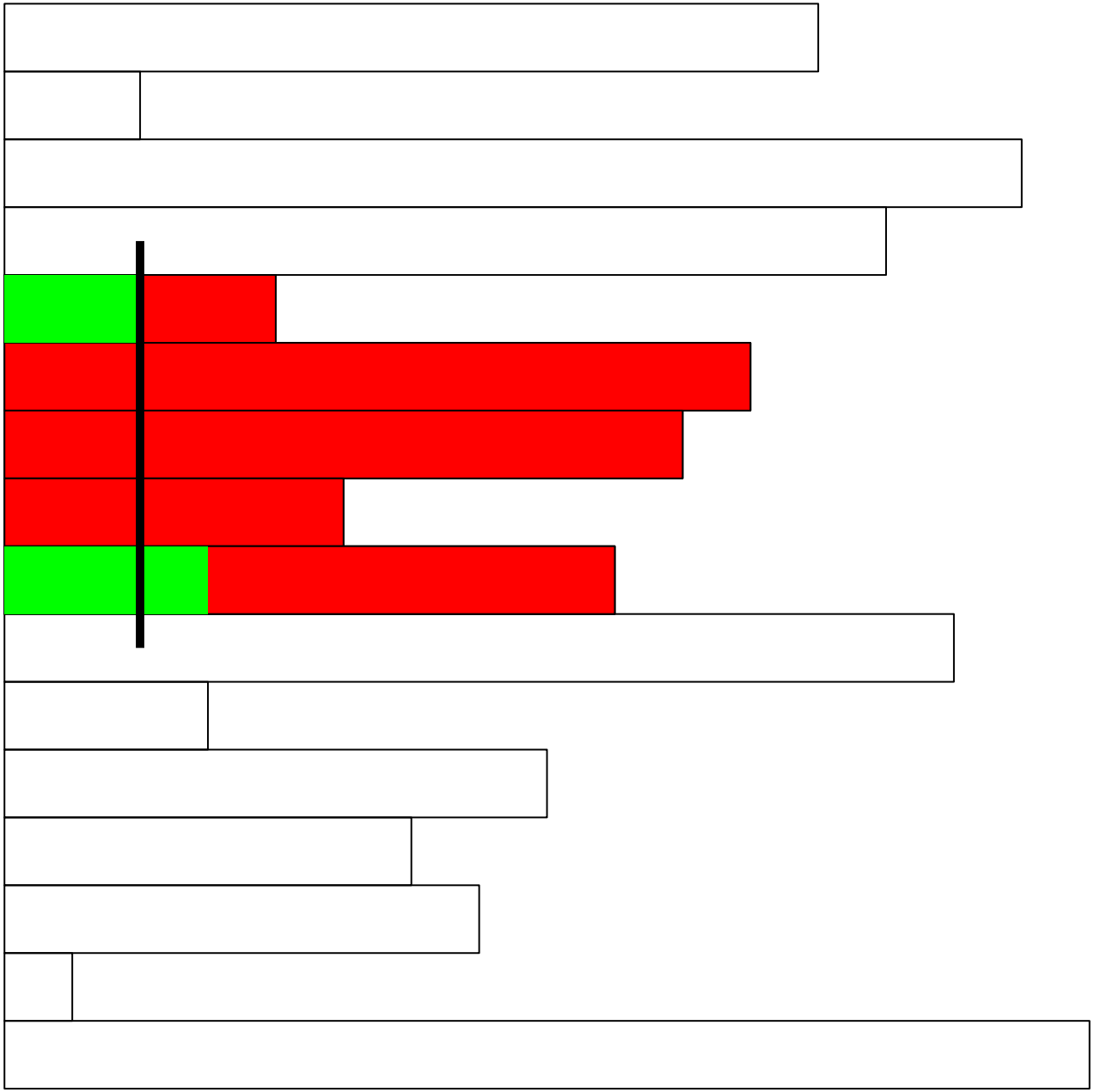
Recall that $\mathrm{lcp}(i,j)$ is the longest common prefix of the suffixes $w[i..n]$ and $w[j..n]$. We know how to compute all $\mathrm{lcp}[i]$, and we observed that computing $\mathrm{lcp}(i,j)$ reduces to the so-called Range Minimum Query on the $\mathrm{lcp}[i]$ array.

For the time being assume that we know how to answer RMQ queries on any array in $\mathcal{O}(1)$ time after linear preprocessing. Then we can compute any $\mathrm{lcp}(i,j)$ in $\mathcal{O}(1)$ time. Can this help us to speed up the binary searching?




L
M
R





Invariant

We maintain a range $[L, R]$ such that the answer is somewhere inside, and we know the longest common prefix of $SA[L]$ and p, and $SA[R]$ and p.

We choose $M \in (L, R)$. Of course we know that the longest common prefix of $SA[M]$ and p is at least as long as the minimum of the two known prefixes, but we can notice even more.
Let ℓ be the longest common prefix of $SA[L]$ and p, and r be the longest common prefix of $SA[R]$ and p. Assume that $\ell \leq r$, the situation is symmetric so the other case is very similar.

Invariant

We maintain a range $[L, R]$ such that the answer is somewhere inside, and we know the longest common prefix of $SA[L]$ and p, and $SA[R]$ and p.

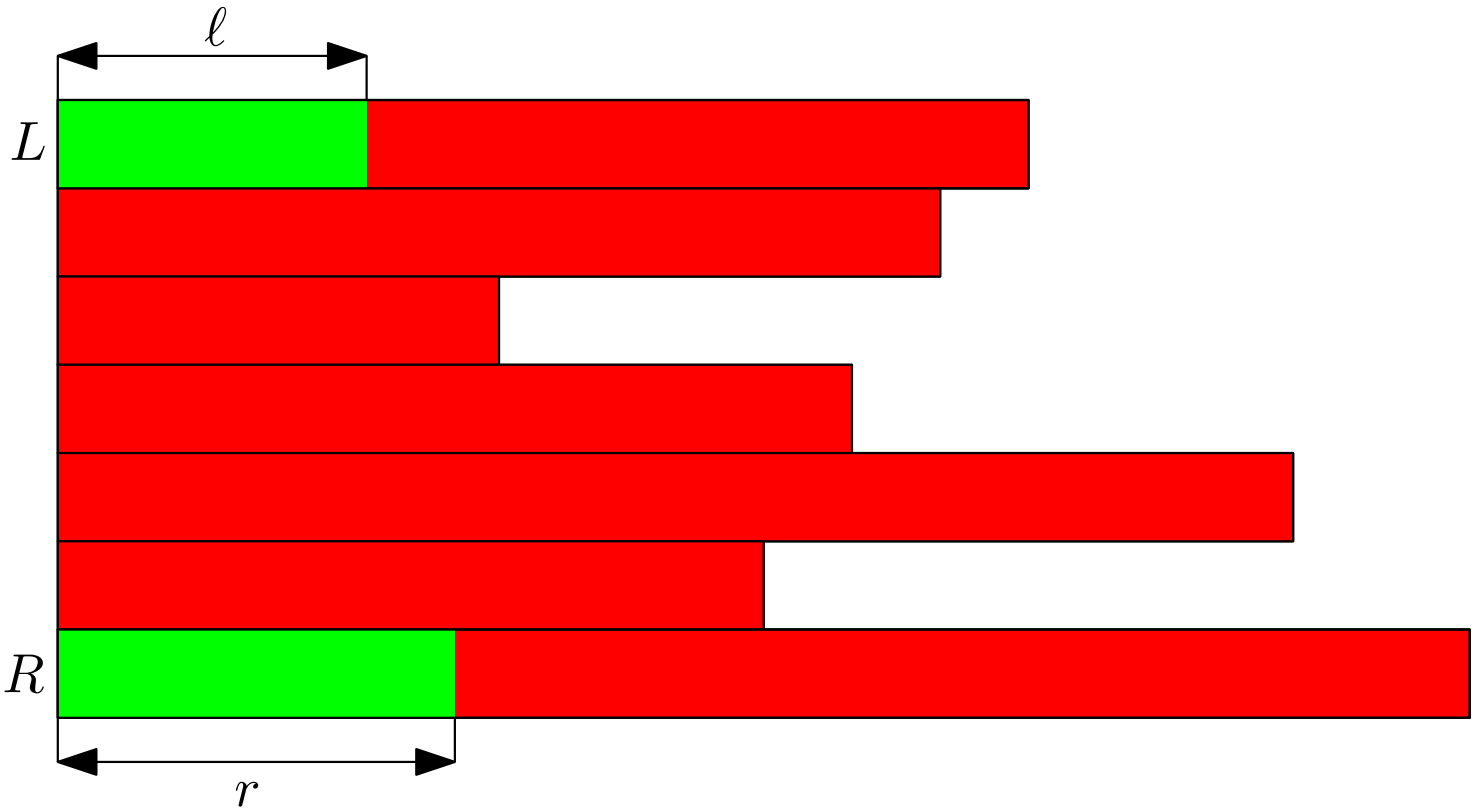
We choose $M \in (L, R)$. Of course we know that the longest common prefix of $SA[M]$ and p is at least as long as the minimum of the two known prefixes, but we can notice even more.

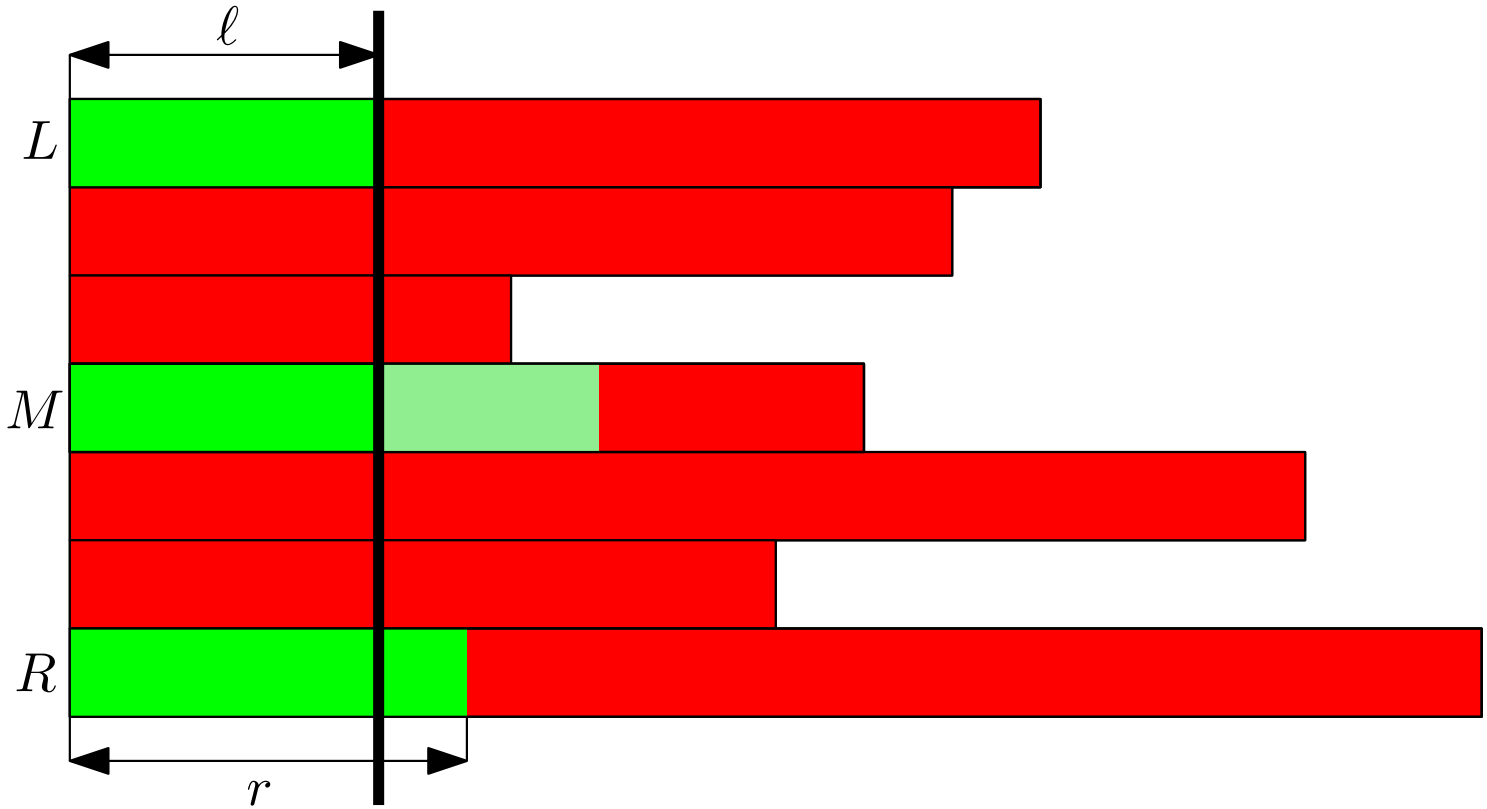
Let ℓ be the longest common prefix of $SA[L]$ and p, and r be the longest common prefix of $SA[R]$ and p. Assume that $\ell \leq r$, the situation is symmetric so the other case is very similar.

Invariant

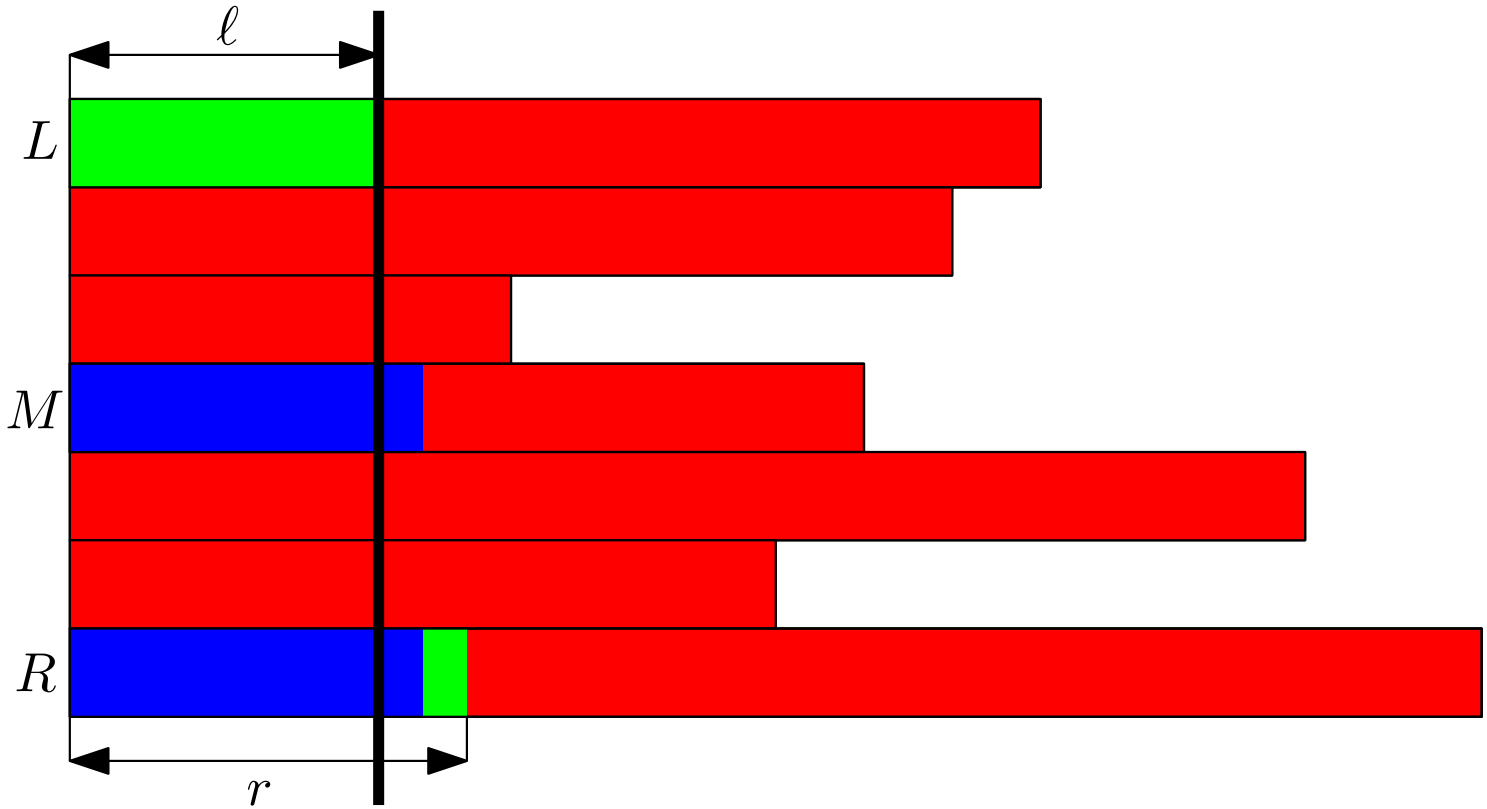
We maintain a range $[L, R]$ such that the answer is somewhere inside, and we know the longest common prefix of $SA[L]$ and p, and $SA[R]$ and p.

We choose $M \in (L, R)$. Of course we know that the longest common prefix of $SA[M]$ and p is at least as long as the minimum of the two known prefixes, but we can notice even more.
Let ℓ be the longest common prefix of $SA[L]$ and p, and r be the longest common prefix of $SA[R]$ and p. Assume that $\ell \leq r$, the situation is symmetric so the other case is very similar.

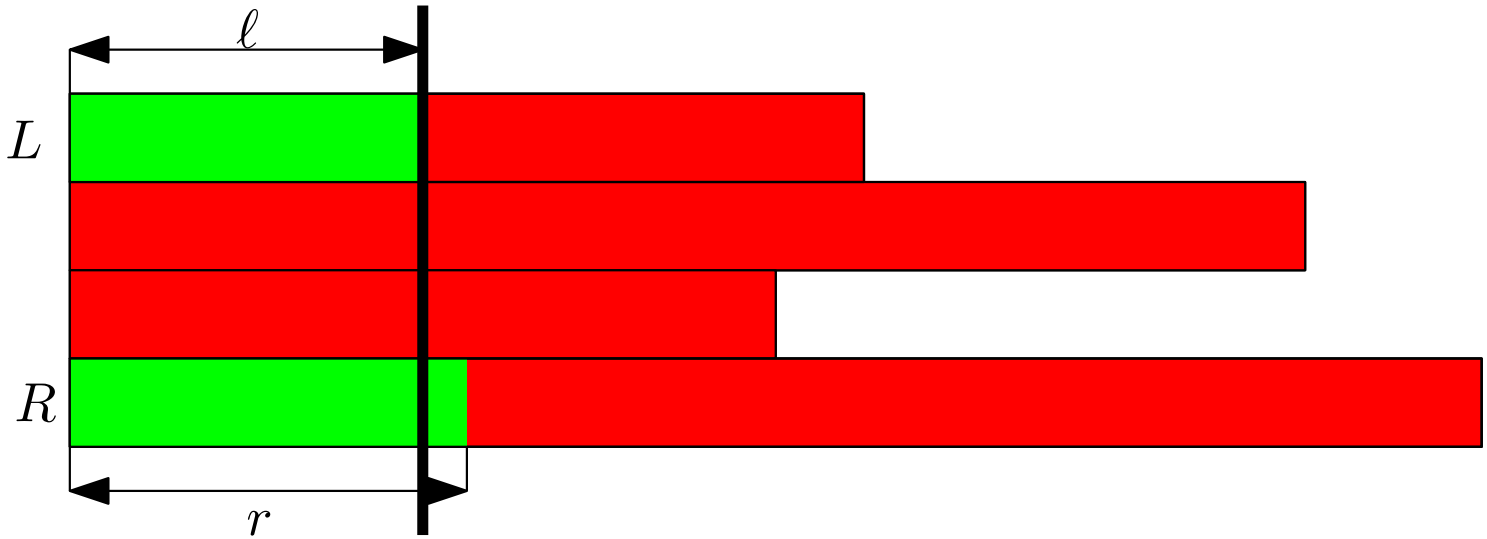

ℓ
L
R
r



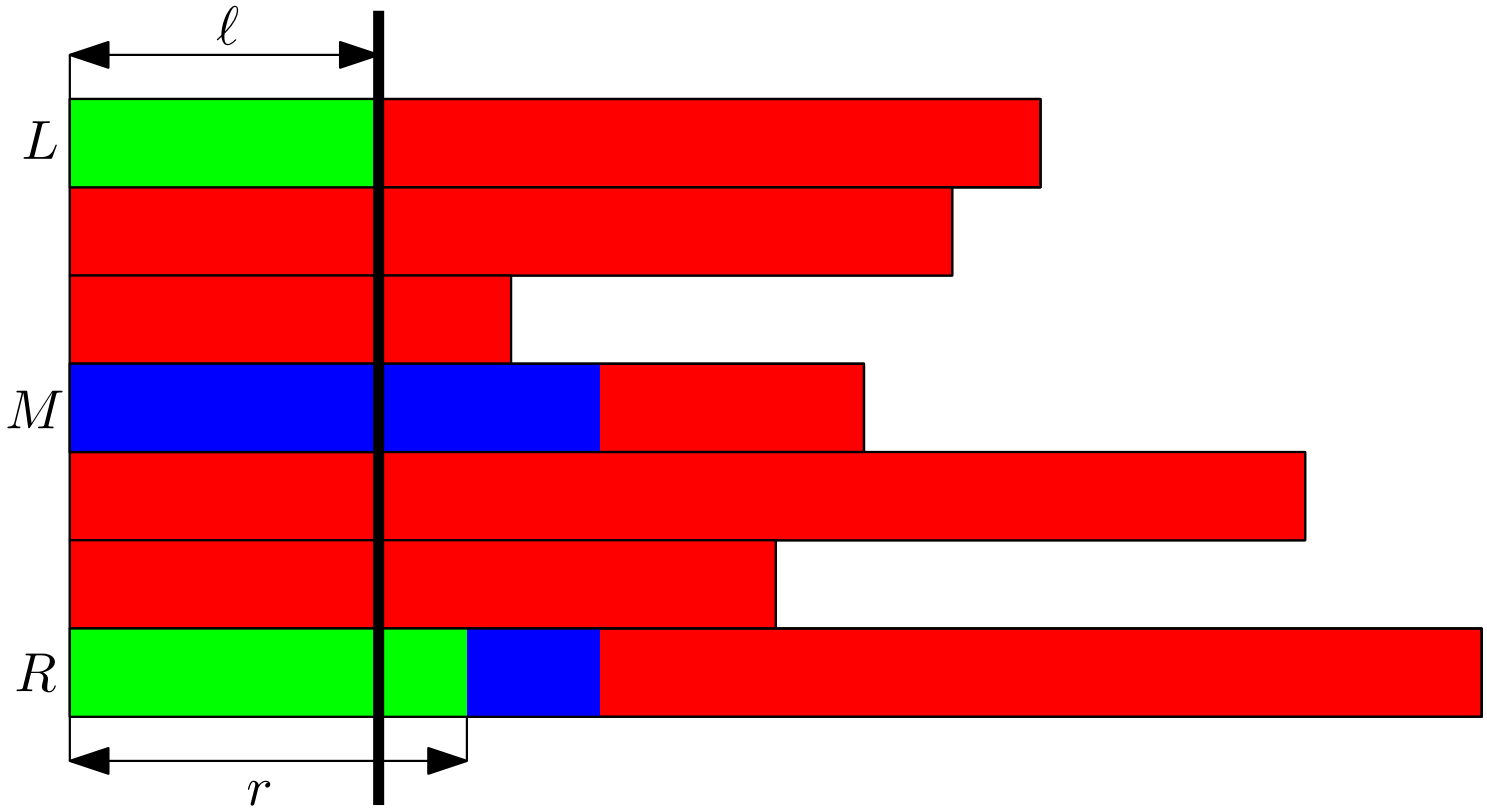
Look at $\mathrm{lcp}(SA[M], SA[R])$.



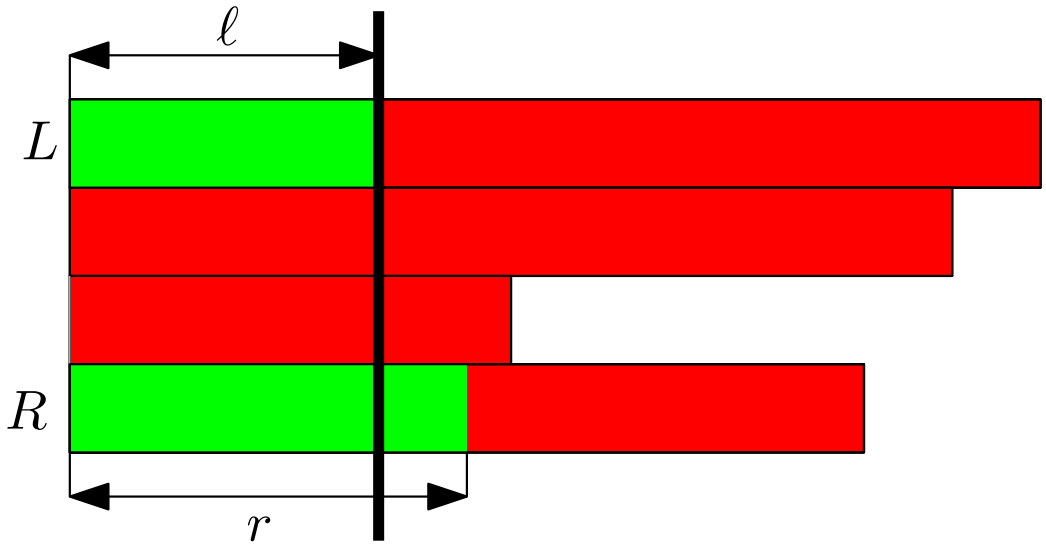
If $\mathrm{lcp}(SA[M], SA[R]) < r$, set $L = M$ and $\ell = \mathrm{lcp}(SA[M], SA[R])$.



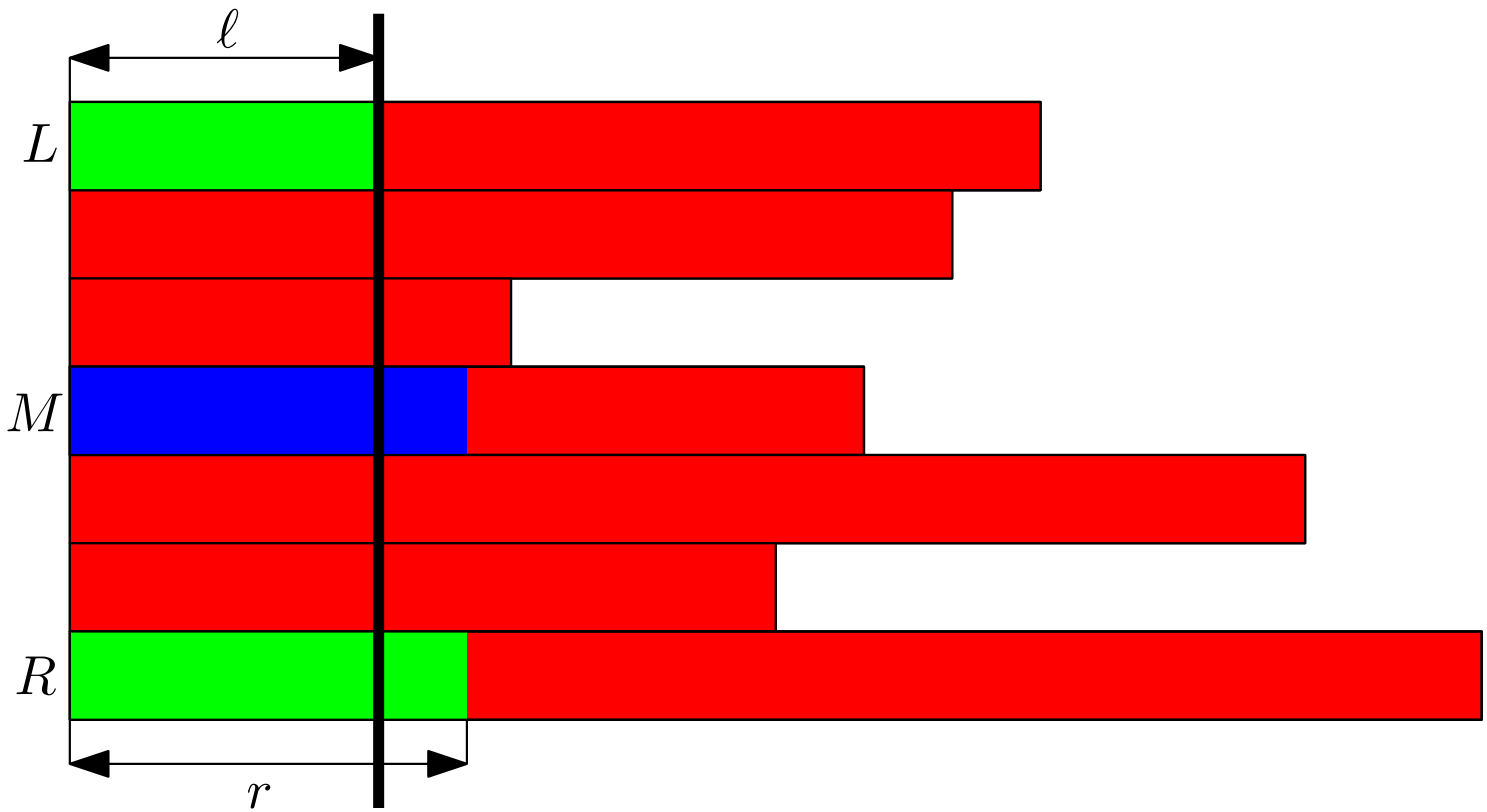
If $\mathrm{lcp}(SA[M], SA[R]) < r$, set $L = M$ and $\ell = \mathrm{lcp}(SA[M], SA[R])$.



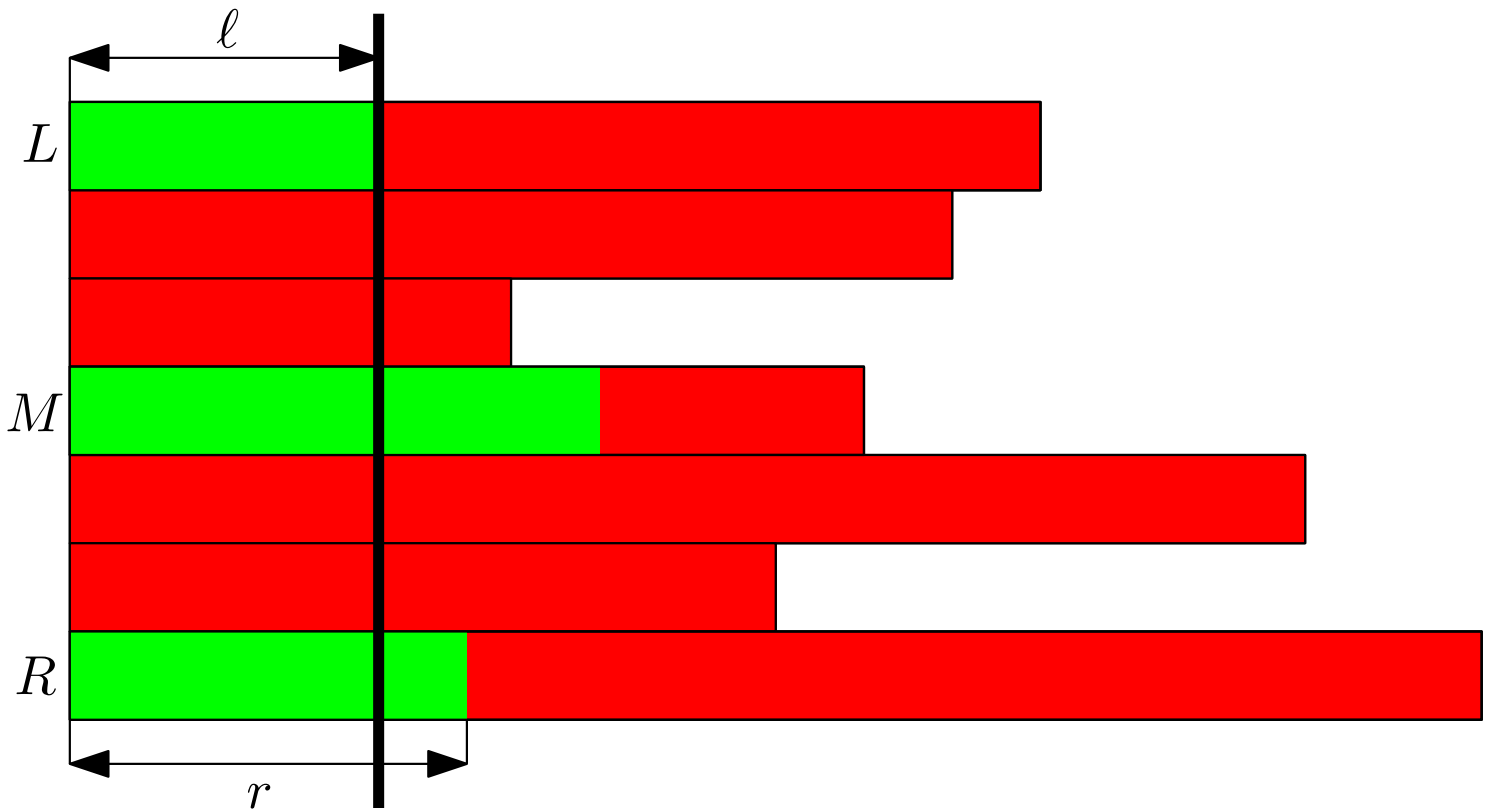
If $\mathrm{lcp}(SA[M], SA[R]) > r$, set $R = M$ and keep old ℓ and r.



If $\mathrm{lcp}(SA[M], SA[R]) > r$, set $R = M$ and keep old ℓ and r.



If $\mathrm{lcp}(SA[M], SA[R]) = r$, compute the longest common prefix of $SA[M]$ and p, but start from the r-th character. Depending on the next character set $L = M$ or $R = M$.



If $\mathrm{lcp}(SA[M], SA[R]) = r$, compute the longest common prefix of $SA[M]$ and p, but start from the r-th character. Depending on the next character set $L = M$ or $R = M$.

Let's look again at the last case. Say that the longest common prefix of $SA[M]$ and p be k. We have two cases:

- the next character of $SA[M]$ is less than $p[k+1]$, then we set $L = M$ and $\ell = k$,
- the next character of $SA[M]$ is greater than $p[k+1]$, then we set $R = M$ and $r = k$.

In both cases we spent just $\mathcal{O}(k - r + 1)$ time computing the longest common prefix.

The value of $\ell + r$ doesn't decrease.

It follows that the sum of $\mathcal{O}(k - r)$ over all steps of the procedure is just $\mathcal{O}(m)$ in the worst possible case. We additionally spent $\mathcal{O}(1)$ time per step to look at $SA[M]$, hence the total complexity is $\mathcal{O}(m + \log n)$.

Let's look again at the last case. Say that the longest common prefix of $SA[M]$ and p be k. We have two cases:

- the next character of $SA[M]$ is less than $p[k+1]$, then we set $L = M$ and $\ell = k$,
- the next character of $SA[M]$ is greater than $p[k+1]$, then we set $R = M$ and $r = k$.

In both cases we spent just $\mathcal{O}(k - r + 1)$ time computing the longest common prefix.

The value of $\ell + r$ doesn't decrease.

It follows that the sum of $\mathcal{O}(k - r)$ over all steps of the procedure is just $\mathcal{O}(m)$ in the worst possible case. We additionally spent $\mathcal{O}(1)$ time per step to look at $SA[M]$, hence the total complexity is $\mathcal{O}(m + \log n)$.

Let's look again at the last case. Say that the longest common prefix of $SA[M]$ and p be k. We have two cases:

- the next character of $SA[M]$ is less than $p[k+1]$, then we set $L = M$ and $\ell = k$,
- the next character of $SA[M]$ is greater than $p[k+1]$, then we set $R = M$ and $r = k$.

In both cases we spent just $\mathcal{O}(k - r + 1)$ time computing the longest common prefix.

The value of $\ell + r$ doesn't decrease.

It follows that the sum of $\mathcal{O}(k - r)$ over all steps of the procedure is just $\mathcal{O}(m)$ in the worst possible case. We additionally spent $\mathcal{O}(1)$ time per step to look at $SA[M]$, hence the total complexity is $\mathcal{O}(m + \log n)$.

Let's look again at the last case. Say that the longest common prefix of $SA[M]$ and p be k. We have two cases:

- the next character of $SA[M]$ is less than $p[k+1]$, then we set $L = M$ and $\ell = k$,
- the next character of $SA[M]$ is greater than $p[k+1]$, then we set $R = M$ and $r = k$.

In both cases we spent just $\mathcal{O}(k - r + 1)$ time computing the longest common prefix.

The value of $\ell + r$ doesn't decrease.

It follows that the sum of $\mathcal{O}(k - r)$ over all steps of the procedure is just $\mathcal{O}(m)$ in the worst possible case. We additionally spent $\mathcal{O}(1)$ time per step to look at $SA[M]$, hence the total complexity is $\mathcal{O}(m + \log n)$.

Questions?