Suffix arrays continued

Paweł Gawrychowski

University of Wrocław

SA and lcp arrays

$w =$ 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

SA and lcp arrays

$w =$ mississippi

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

Why?

Recall that we used $\mathrm{lcp}(i, j)$ queries to decrease complexity of pattern matching from $\mathcal{O}(m \log n)$ to $\mathcal{O}(m + \log n)$.

Lemma

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

Why?

Recall that we used $\mathrm{lcp}(i, j)$ queries to decrease complexity of pattern matching from $\mathcal{O}(m \log n)$ to $\mathcal{O}(m + \log n)$.

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.

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

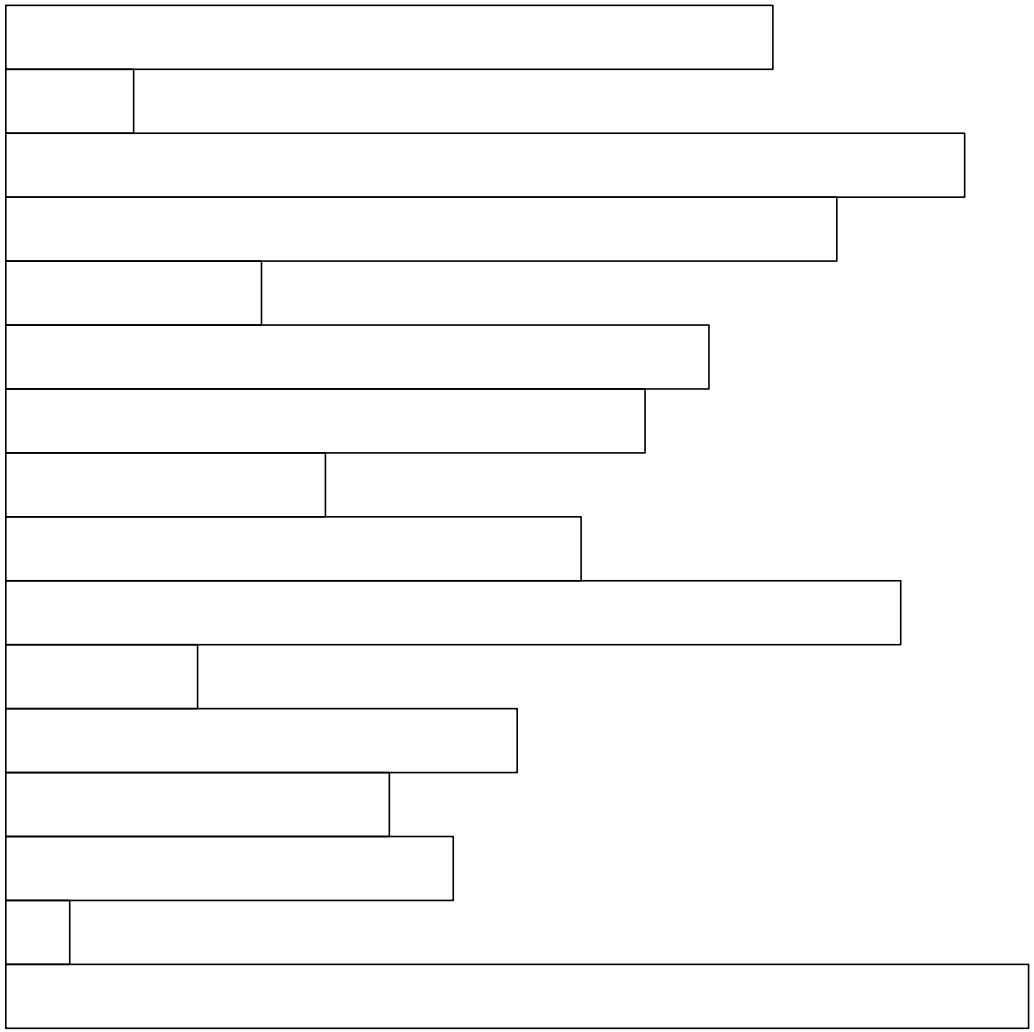
For any suffix $w[i..n]$, $\mathrm{lcp}[SA^{-1}[i]] - 1 \leq \mathrm{lcp}[SA^{-1}[i+1]]$, assuming $i < n$ and neither $SA^{-1}[i]$ nor $SA^{-1}[i+1]$ is the first element of the suffix array.

Kasai et al. 2001

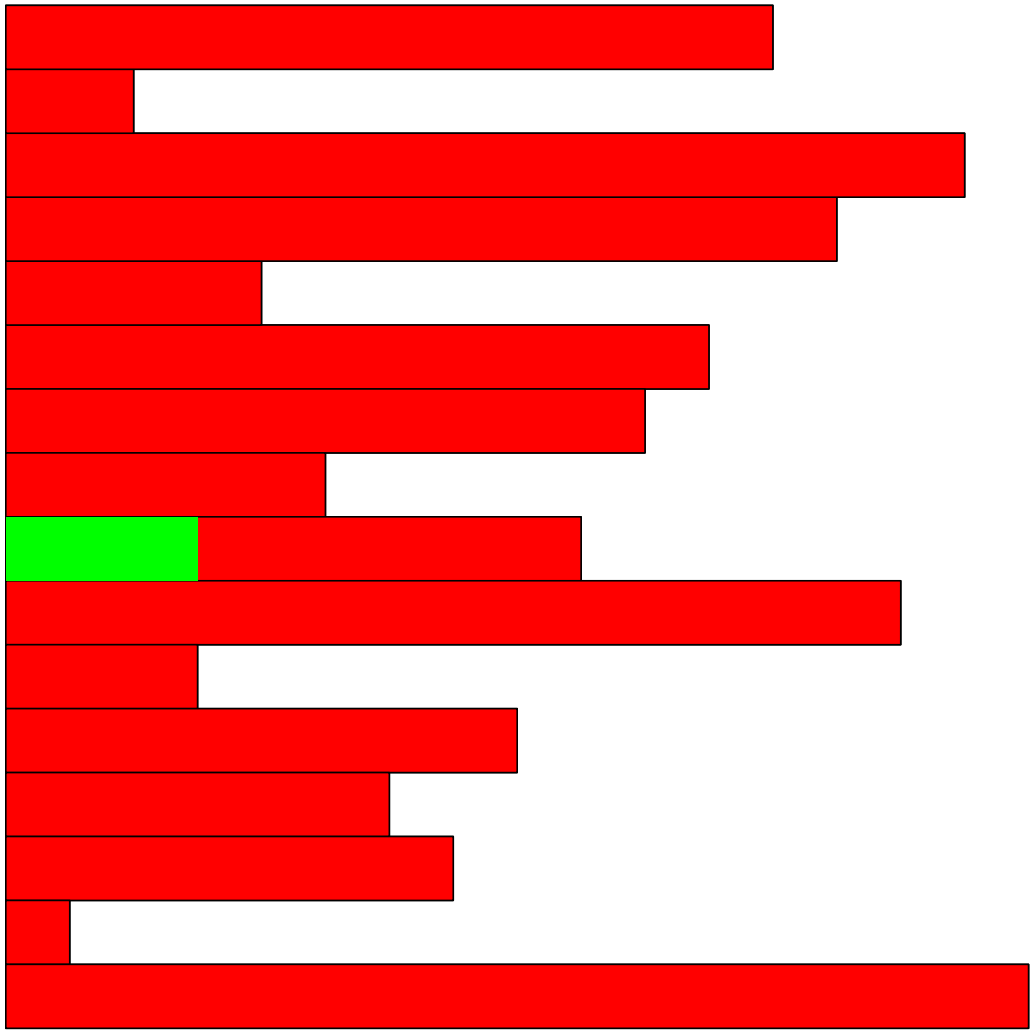
All lcp can be computed in (amortized) $\mathcal{O}(1)$ time per entry.

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.

Recall that we binary search over SA. In every step, we might need $\mathrm{lcp}(SA[L], SA[M])$ or $\mathrm{lcp}(SA[M], SA[R])$.

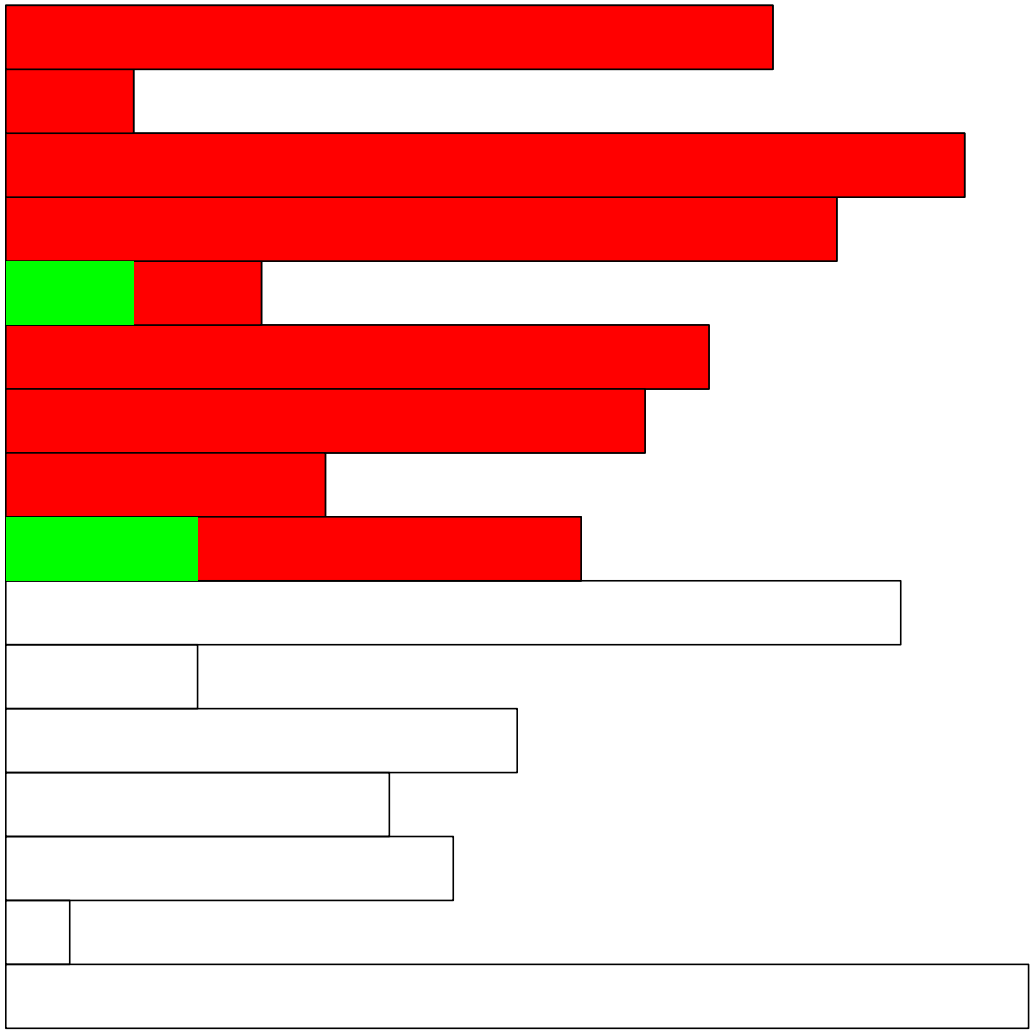


Recall that we binary search over SA. In every step, we might need $\mathrm{lcp}(SA[L], SA[M])$ or $\mathrm{lcp}(SA[M], SA[R])$.



Recall that we binary search over SA. In every step, we might need $\mathrm{lcp}(SA[L], SA[M])$ or $\mathrm{lcp}(SA[M], SA[R])$.

Recall that we binary search over SA. In every step, we might need $\mathrm{lcp}(SA[L], SA[M])$ or $\mathrm{lcp}(SA[M], SA[R])$.



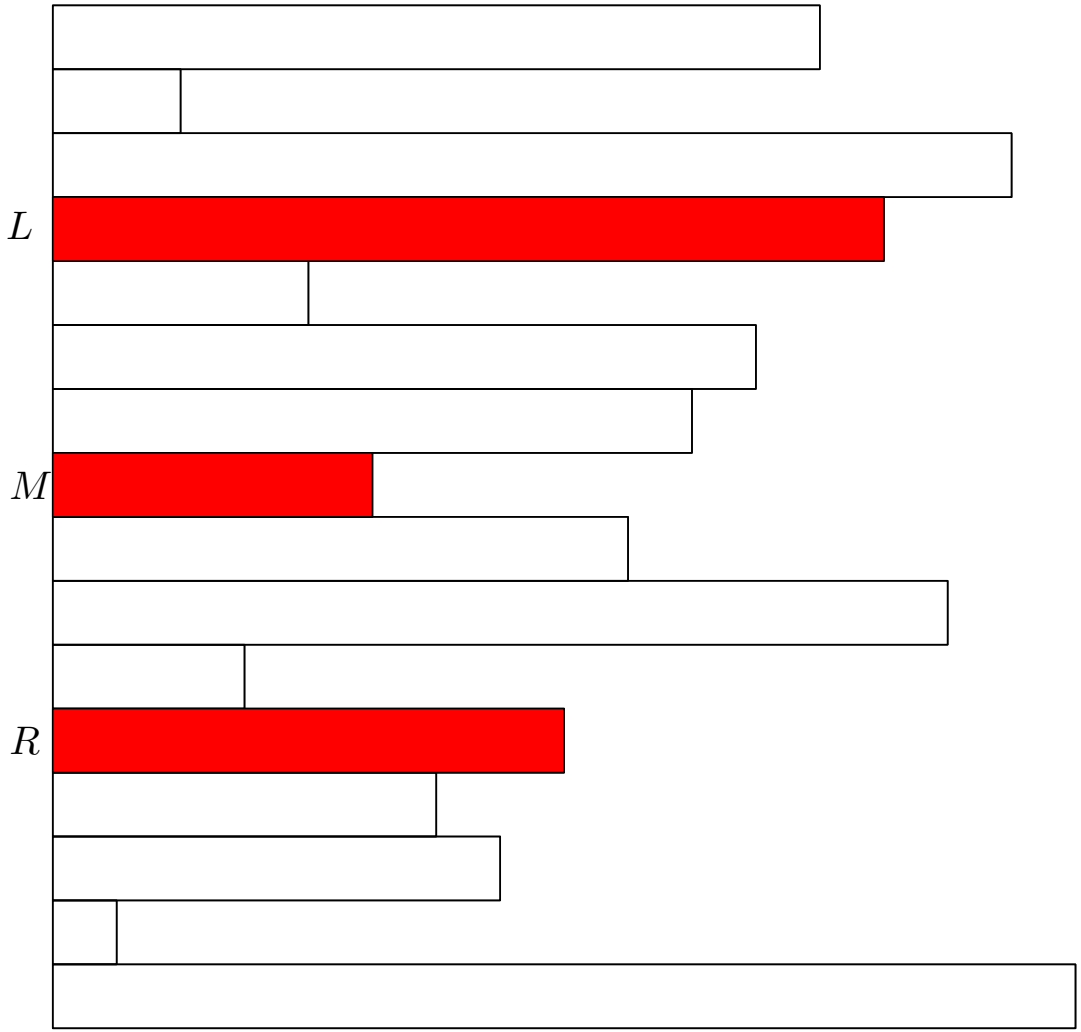
Recall that we binary search over SA. In every step, we might need $\mathrm{lcp}(SA[L], SA[M])$ or $\mathrm{lcp}(SA[M], SA[R])$.

Recall that we binary search over SA. In every step, we might need $\mathrm{lcp}(SA[L], SA[M])$ or $\mathrm{lcp}(SA[M], SA[R])$.

Recall that we binary search over SA. In every step, we might need $\mathrm{lcp}(SA[L], SA[M])$ or $\mathrm{lcp}(SA[M], SA[R])$.

Recall that we binary search over SA. In every step, we might need $\mathrm{lcp}(SA[L], SA[M])$ or $\mathrm{lcp}(SA[M], SA[R])$.

Recall that we binary search over SA. In every step, we might need $\mathrm{lcp}(SA[L], SA[M])$ or $\mathrm{lcp}(SA[M], SA[R])$.

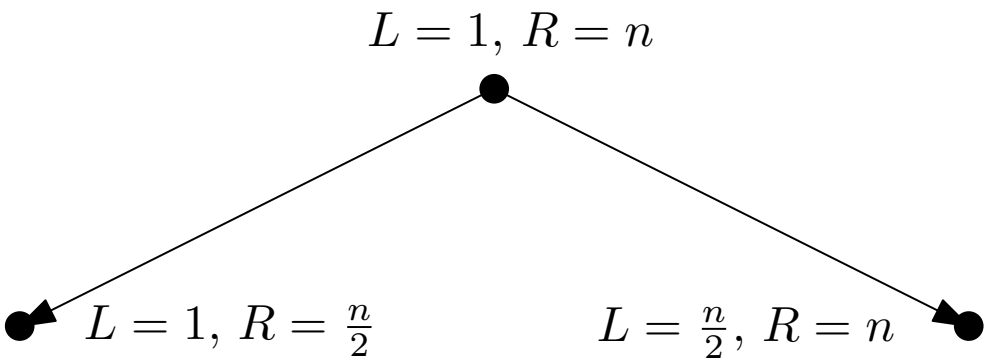


We assumed that computing any $\mathrm{lcp}(i, j)$ takes $\mathcal{O}(1)$ time. While it can be done (as we will soon see), that's an overkill. Do we really need to compute any such value?

$$L = 1,\ R = n$$

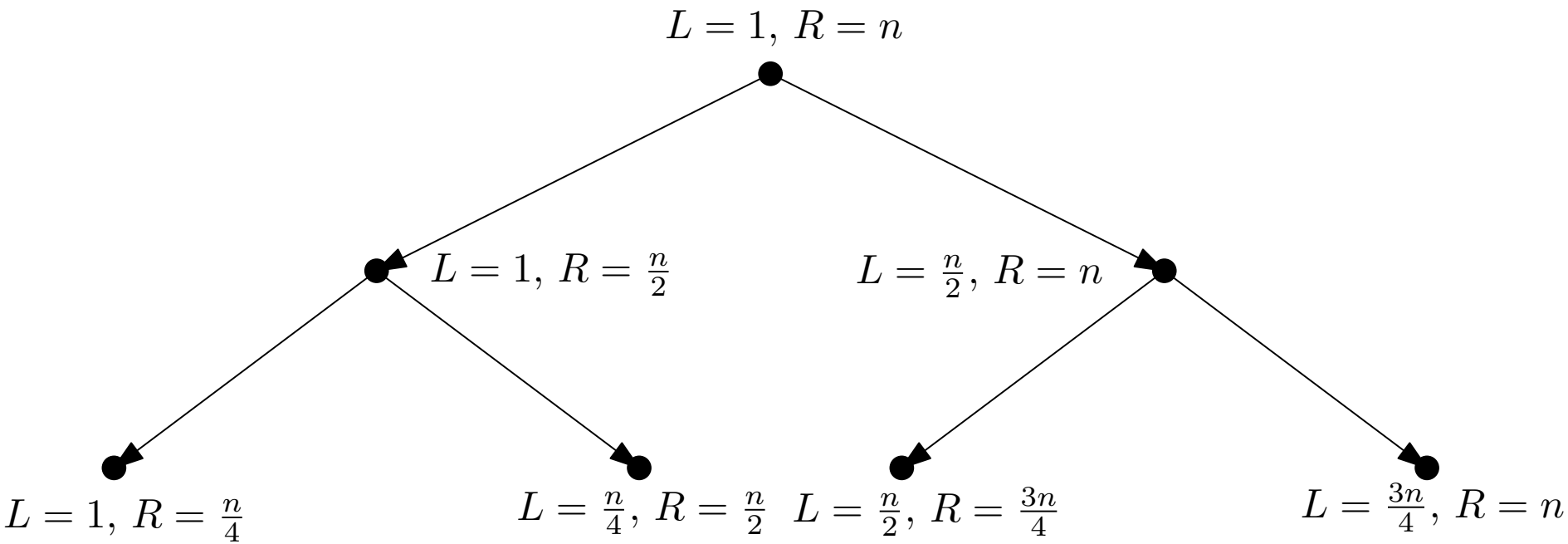
Each node of the recursion tree generates just two values $\mathrm{lcp}(SA[L], SA[M])$ and $\mathrm{lcp}(SA[M], SA[R])$ to be computed. Hence we have just $\mathcal{O}(n)$ values in total!

We assumed that computing any $\mathrm{lcp}(i, j)$ takes $\mathcal{O}(1)$ time. While it can be done (as we will soon see), that's an overkill. Do we really need to compute any such value?



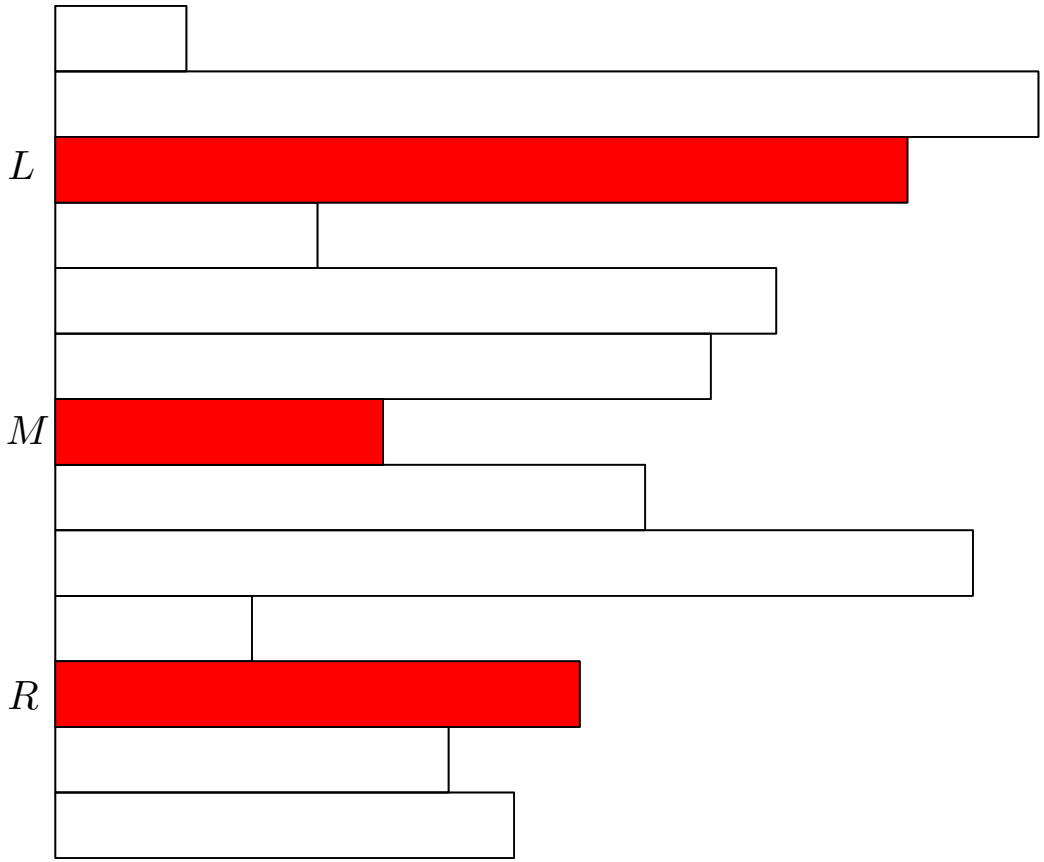
Each node of the recursion tree generates just two values $\mathrm{lcp}(SA[L], SA[M])$ and $\mathrm{lcp}(SA[M], SA[R])$ to be computed. Hence we have just $\mathcal{O}(n)$ values in total!

We assumed that computing any $\mathrm{lcp}(i, j)$ takes $\mathcal{O}(1)$ time. While it can be done (as we will soon see), that's an overkill. Do we really need to compute any such value?



Each node of the recursion tree generates just two values $\mathrm{lcp}(SA[L], SA[M])$ and $\mathrm{lcp}(SA[M], SA[R])$ to be computed. Hence we have just $\mathcal{O}(n)$ values in total!

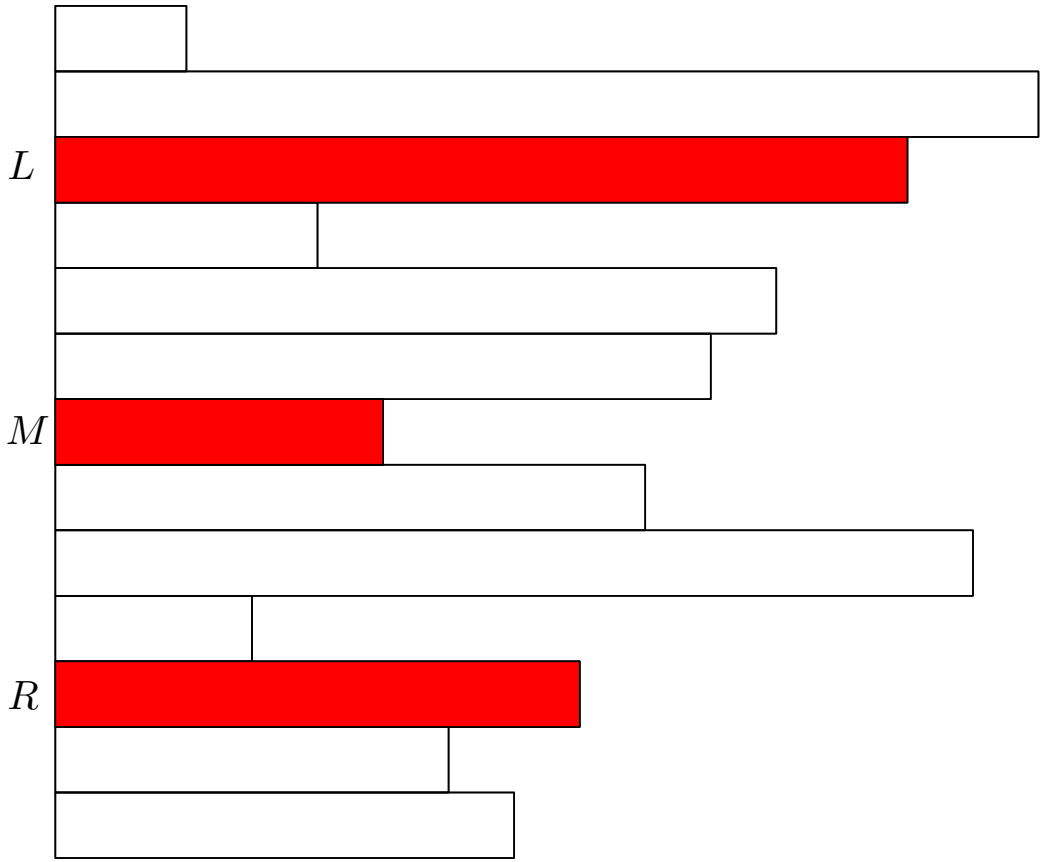
All those values can be actually computed in $\mathcal{O}(n)$ time in a bottom-top manner.



Lemma

$\mathrm{lcp}(SA[L], SA[R]) = \min(\mathrm{lcp}(SA[L], SA[M]), \mathrm{lcp}(SA[M], SA[R]))$

All those values can be actually computed in $\mathcal{O}(n)$ time in a bottom-top manner.



Lemma

$$\mathrm{lcp}(SA[L], SA[R]) = \min(\mathrm{lcp}(SA[L], SA[M]), \mathrm{lcp}(SA[M], SA[R]))$$

Even though we showed that storing just $2n$ values of $\mathrm{lcp}(i,j)$ allows us to execute the binary search efficiently, being able to answer any $\mathrm{lcp}(i,j)$ would be great (we will see why later, maybe). Recall that we were able to reduce the question to the so-called RMQ problem.

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.

First observe that answering any query in $\mathcal{O}(1)$ is trivial if we allow $\mathcal{O}(n^2)$ time and space preprocessing.

Even though we showed that storing just $2n$ values of $\mathrm{lcp}(i,j)$ allows us to execute the binary search efficiently, being able to answer any $\mathrm{lcp}(i,j)$ would be great (we will see why later, maybe). Recall that we were able to reduce the question to the so-called RMQ problem.

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.

First observe that answering any query in $\mathcal{O}(1)$ is trivial if we allow $\mathcal{O}(n^2)$ time and space preprocessing.

Even though we showed that storing just $2n$ values of $\mathrm{lcp}(i,j)$ allows us to execute the binary search efficiently, being able to answer any $\mathrm{lcp}(i,j)$ would be great (we will see why later, maybe). Recall that we were able to reduce the question to the so-called RMQ problem.

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.

First observe that answering any query in $\mathcal{O}(1)$ is trivial if we allow $\mathcal{O}(n^2)$ time and space preprocessing.

Lemma

RMQ can be solved in $\mathcal{O}(1)$ time after $\mathcal{O}(n \log n)$ time and space preprocessing.

To prove the lemma, we will (again) apply the simple-yet-powerful doubling technique. For each $k = 0, 1, \ldots, \log n$ construct a table B_k.

$$B_k[i] = \min\{A[i], A[i+1], A[i+2], \ldots, A[i+2^k-1]\}$$

How? Well, $B_0[i] = A[i]$, and $B_{k+1}[i] = \min(B_k[i], B_k[i+2^k])$. Hence we can easily answer a query concerning a fragment of length that is a power of 2. But, unfortunately, not all numbers are powers of 2...

Lemma

RMQ can be solved in $\mathcal{O}(1)$ time after $\mathcal{O}(n \log n)$ time and space preprocessing.

To prove the lemma, we will (again) apply the simple-yet-powerful doubling technique. For each $k = 0, 1, \ldots, \log n$ construct a table B_k.

$$B_k[i] = \min\{A[i], A[i+1], A[i+2], \ldots, A[i+2^k-1]\}$$

How? Well, $B_0[i] = A[i]$, and $B_{k+1}[i] = \min(B_k[i], B_k[i+2^k])$. Hence we can easily answer a query concerning a fragment of length that is a power of 2. But, unfortunately, not all numbers are powers of 2...

Lemma

RMQ can be solved in $\mathcal{O}(1)$ time after $\mathcal{O}(n \log n)$ time and space preprocessing.

To prove the lemma, we will (again) apply the simple-yet-powerful doubling technique. For each $k = 0, 1, \ldots, \log n$ construct a table B_k.

$$B_k[i] = \min\{A[i], A[i+1], A[i+2], \ldots, A[i+2^k-1]\}$$

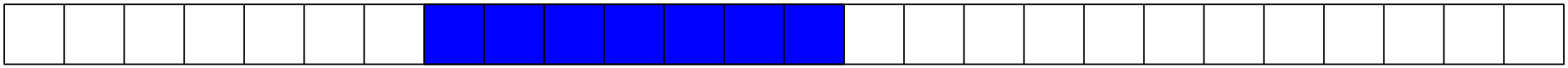
How? Well, $B_0[i] = A[i]$, and $B_{k+1}[i] = \min(B_k[i], B_k[i+2^k])$. Hence we can easily answer a query concerning a fragment of length that is a power of 2. But, unfortunately, not all numbers are powers of 2...

...or are they?

Answering a query concerning a range [i,j]

To figure out which two power-of-two queries should be asked, compute $k = \lfloor \log j - i + 1 \rfloor$. Then return $\min(B_k[i], B_k[j - 2^k + 1])$.

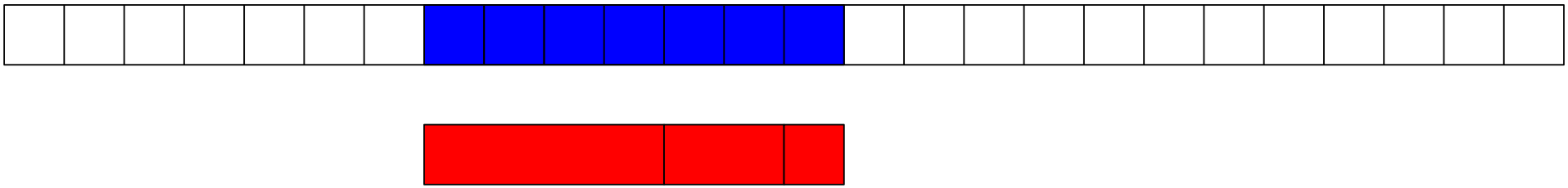
...or are they?



Answering a query concerning a range [i,j]

To figure out which two power-of-two queries should be asked, compute $k = \lfloor \log j - i + 1 \rfloor$. Then return $\min(B_k[i], B_k[j - 2^k + 1])$.

...or are they?

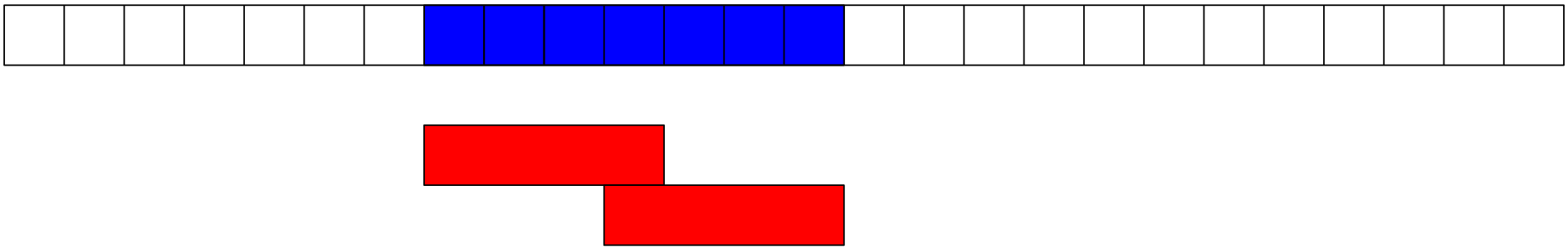


Any query can be split into at most $\log n$ power-of-two queries.

Answering a query concerning a range [i,j]

To figure out which two power-of-two queries should be asked, compute $k = \lfloor \log j - i + 1 \rfloor$. Then return $\min(B_k[i], B_k[j - 2^k + 1])$.

...or are they?

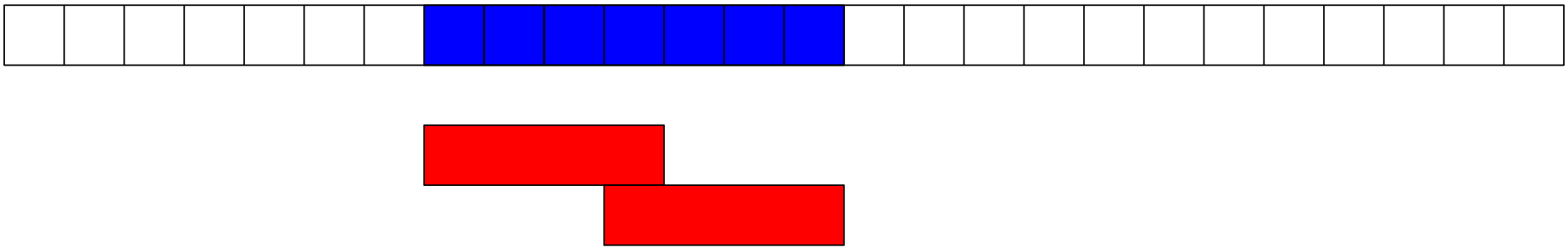


Any query can be covered with 2 power-of-two queries.

Answering a query concerning a range [i,j]

To figure out which two power-of-two queries should be asked, compute $k = \lfloor \log j - i + 1 \rfloor$. Then return $\min(B_k[i], B_k[j - 2^k + 1])$.

...or are they?



Any query can be covered with 2 power-of-two queries.

Answering a query concerning a range [i,j]

To figure out which two power-of-two queries should be asked, compute $k = \lfloor \log j - i + 1 \rfloor$. Then return $\min(B_k[i], B_k[j - 2^k + 1])$.

Lemma

RMQ can be solved in $\mathcal{O}(\log n)$ time after $\mathcal{O}(n)$ time and space preprocessing.

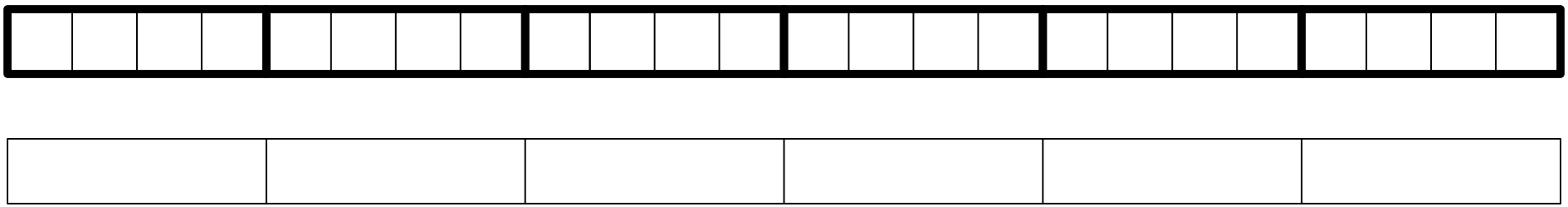
We apply another simple-yet-powerful technique: micro-macro decomposition. Chop the input array into blocks of length $b = \log n$.

Construct a new array A', where $A'[i] = \min\{A[ib+1], A[ib+2], \ldots, A[(i+1)b]\}$. Build the previously described structure for A'.

Lemma

RMQ can be solved in $\mathcal{O}(\log n)$ time after $\mathcal{O}(n)$ time and space preprocessing.

We apply another simple-yet-powerful technique: micro-macro decomposition. Chop the input array into blocks of length $b = \log n$.

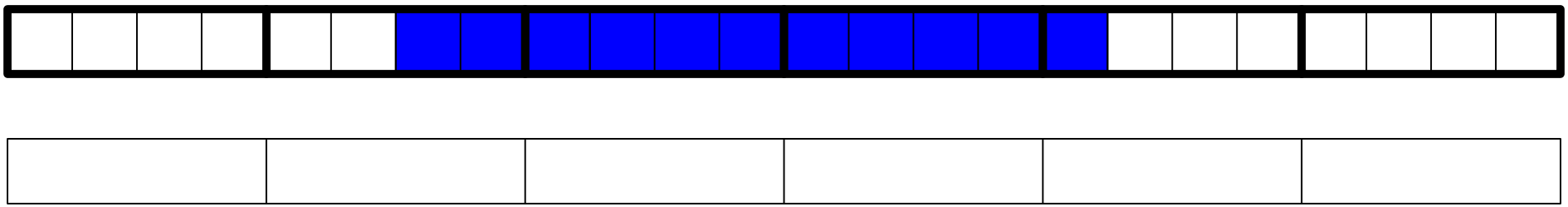


Construct a new array A', where $A'[i] = \min\{A[ib+1], A[ib+2], \ldots, A[(i+1)b]\}$. Build the previously described structure for A'.

Lemma

RMQ can be solved in $\mathcal{O}(\log n)$ time after $\mathcal{O}(n)$ time and space preprocessing.

We apply another simple-yet-powerful technique: micro-macro decomposition. Chop the input array into blocks of length $b = \log n$.

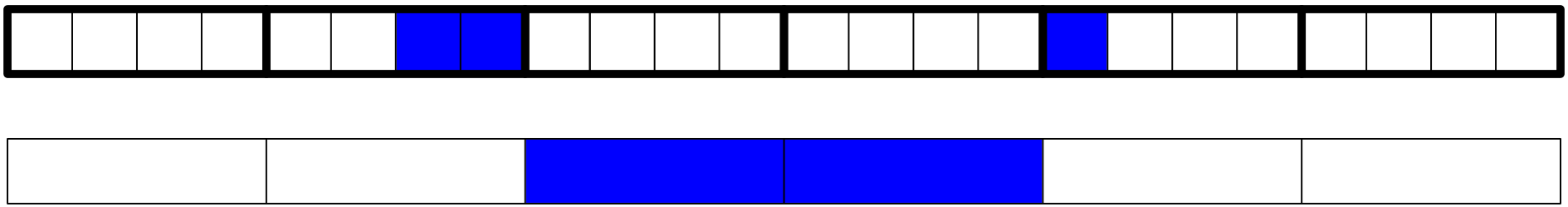


Construct a new array A', where $A'[i] = \min\{A[ib+1], A[ib+2], \ldots, A[(i+1)b]\}$. Build the previously described structure for A'.

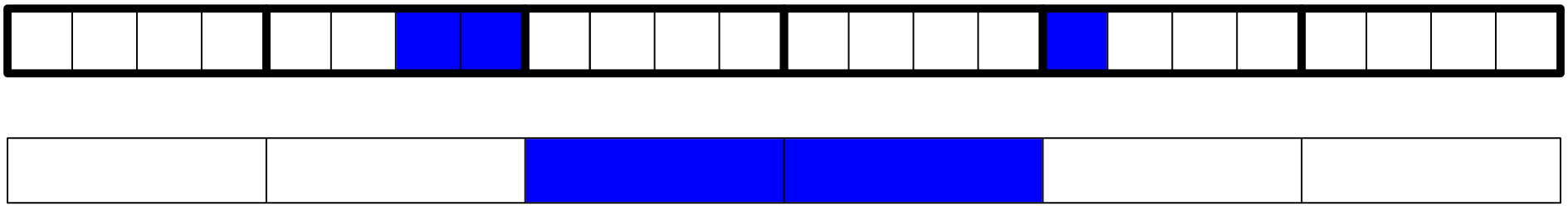
Lemma

RMQ can be solved in $\mathcal{O}(\log n)$ time after $\mathcal{O}(n)$ time and space preprocessing.

We apply another simple-yet-powerful technique: micro-macro decomposition. Chop the input array into blocks of length $b = \log n$.



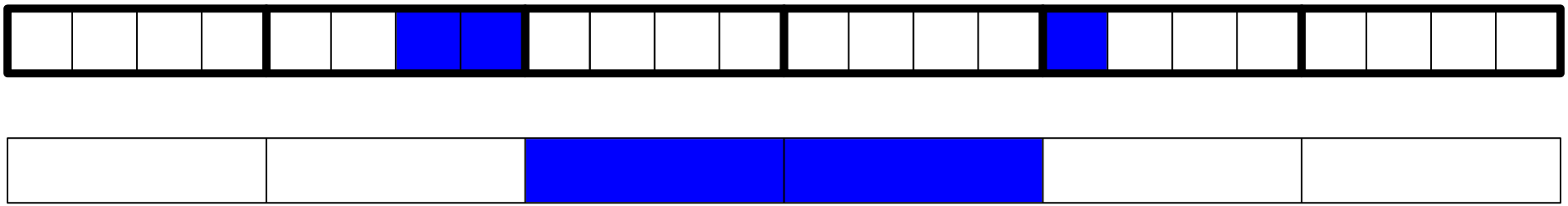
Construct a new array A', where $A'[i] = \min\{A[ib+1], A[ib+2], \ldots, A[(i+1)b]\}$. Build the previously described structure for A'.



For each block, precompute the maximum in each prefix and each suffix, which takes just $\mathcal{O}(n)$ time and space. Then, using the structure built for A', we can answer any query in $\mathcal{O}(1)$ time.

Unfortunately, life is not that simple.

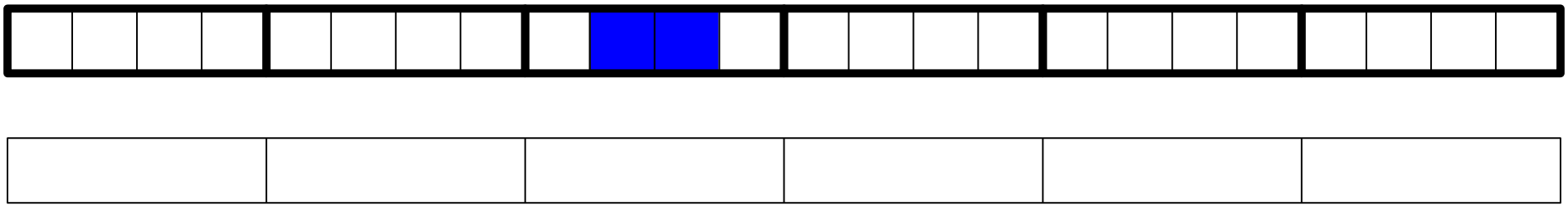
But the only case when we cannot answer a query in $\mathcal{O}(1)$ time is when the range is strictly inside a single block. Revert to the naive one-by-one computation!



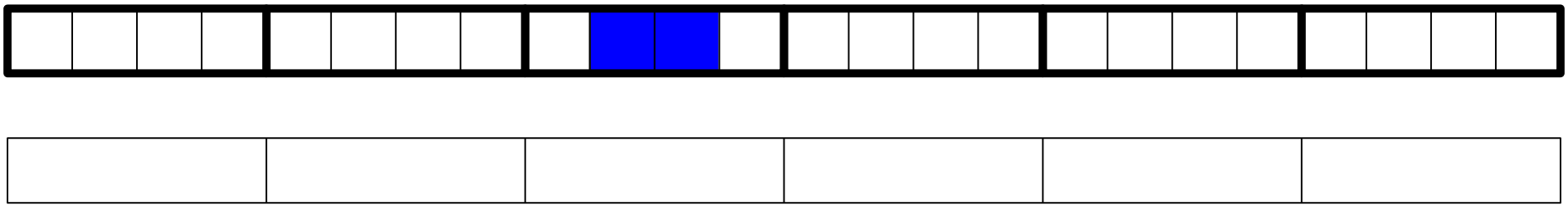
For each block, precompute the maximum in each prefix and each suffix, which takes just $\mathcal{O}(n)$ time and space. Then, using the structure built for A', we can answer any query in $\mathcal{O}(1)$ time.

Unfortunately, life is not that simple.

But the only case when we cannot answer a query in $\mathcal{O}(1)$ time is when the range is strictly inside a single block. Revert to the naive one-by-one computation!



For each block, precompute the maximum in each prefix and each suffix, which takes just $\mathcal{O}(n)$ time and space. Then, using the structure built for A', we can answer any query in $\mathcal{O}(1)$ time.
Unfortunately, life is not that simple.
But the only case when we cannot answer a query in $\mathcal{O}(1)$ time is when the range is strictly inside a single block. Revert to the naive one-by-one computation!



For each block, precompute the maximum in each prefix and each suffix, which takes just $\mathcal{O}(n)$ time and space. Then, using the structure built for A', we can answer any query in $\mathcal{O}(1)$ time.
Unfortunately, life is not that simple.
But the only case when we cannot answer a query in $\mathcal{O}(1)$ time is when the range is strictly inside a single block. Revert to the naive one-by-one computation!

OK, but we promised the best of both worlds: $\mathcal{O}(1)$ query and $\mathcal{O}(n)$ space.

Lemma

RMQ can be solved in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$ time and space preprocessing.

We "only" have to deal with the strictly-inside-a-block case. We will show how to do this for a restricted case where $|A[i+1] - A[i]| \leq 1$.

The general case can be always reduced to the restricted case: given an array $A[1..n]$ we can construct a restricted array $A'[1..2n]$ such that answering RMQ on A reduces to answering RMQ on A'.

OK, but we promised the best of both worlds: $\mathcal{O}(1)$ query and $\mathcal{O}(n)$ space.

Lemma

RMQ can be solved in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$ time and space preprocessing.

We "only" have to deal with the strictly-inside-a-block case. We will show how to do this for a restricted case where $|A[i+1] - A[i]| \leq 1$.

The general case can be always reduced to the restricted case: given an array $A[1..n]$ we can construct a restricted array $A'[1..2n]$ such that answering RMQ on A reduces to answering RMQ on A'.

OK, but we promised the best of both worlds: $\mathcal{O}(1)$ query and $\mathcal{O}(n)$ space.

Lemma

RMQ can be solved in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$ time and space preprocessing.

We "only" have to deal with the strictly-inside-a-block case. We will show how to do this for a restricted case where $|A[i+1] - A[i]| \leq 1$.

The general case can be always reduced to the restricted case: given an array $A[1..n]$ we can construct a restricted array $A'[1..2n]$ such that answering RMQ on A reduces to answering RMQ on A'.

The exact values of the elements don't matter that much. So, for each block we compute its type, which is the sequence of differences $A[i+1] - A[i]$. Additionally, for each such sequence we precompute the answers to all possible $\binom{b}{2}$ queries. The answer is the position of the element with the smallest value.

How much space do we need for that?

$3^{b-1}\binom{b}{2} = \mathcal{O}(3^b b^2)$.

As long as $b \leq 0.001 \log n$, this is **small**, or $o(n)$. Then to answer a query strictly inside a block, we look at its type, retrieve the precomputed answer, and then return the value at the corresponding position in A, all in $\mathcal{O}(1)$ time.

The exact values of the elements don't matter that much. So, for each block we compute its type, which is the sequence of differences $A[i+1] - A[i]$. Additionally, for each such sequence we precompute the answers to all possible $\binom{b}{2}$ queries. The answer is the position of the element with the smallest value.

How much space do we need for that?

$3^{b-1}\binom{b}{2} = \mathcal{O}(3^b b^2)$.

As long as $b \leq 0.001 \log n$, this is **small**, or $o(n)$. Then to answer a query strictly inside a block, we look at its type, retrieve the precomputed answer, and then return the value at the corresponding position in A, all in $\mathcal{O}(1)$ time.

The exact values of the elements don't matter that much. So, for each block we compute its type, which is the sequence of differences $A[i+1] - A[i]$. Additionally, for each such sequence we precompute the answers to all possible $\binom{b}{2}$ queries. The answer is the position of the element with the smallest value.

How much space do we need for that?

$3^{b-1}\binom{b}{2} = \mathcal{O}(3^b b^2)$.

As long as $b \leq 0.001 \log n$, this is **small**, or $o(n)$. Then to answer a query strictly inside a block, we look at its type, retrieve the precomputed answer, and then return the value at the corresponding position in A, all in $\mathcal{O}(1)$ time.

Now we will see how to use suffix array to compress data. There different lossless compression methods, but most of the modern applications are using on of the two.

Burrows-Wheeler transform

Based on the suffix array. Not very nice in theory, but useful in practice. Wait 30 minutes!

Lempel-Ziv

If your data contains the same fragment again and again, writing it down multiple times doesn't make much sense. Maybe we could do better? Let's try!

Now we will see how to use suffix array to compress data. There different lossless compression methods, but most of the modern applications are using on of the two.

Burrows-Wheeler transform

Based on the suffix array. Not very nice in theory, but useful in practice. Wait 30 minutes!

Lempel-Ziv

If your data contains the same fragment again and again, writing it down multiple times doesn't make much sense. Maybe we could do better? Let's try!

Now we will see how to use suffix array to compress data. There different lossless compression methods, but most of the modern applications are using on of the two.

Burrows-Wheeler transform

Based on the suffix array. Not very nice in theory, but useful in practice. Wait 30 minutes!

Lempel-Ziv

If your data contains the same fragment again and again, writing it down multiple times doesn't make much sense. Maybe we could do better? Let's try!

Lempel-Ziv based compression schemes

Text t is partitioned into a number of disjoint blocks $b_1 b_2 \ldots b_n$. Each block is defined in terms of the blocks on the left.

What "defined" exactly means depends on the exact variant. The most common are:

- **LZ77** new block b_i is a subword of $b_1 b_2 \ldots b_{i-1}$ concatenated with exactly one character, `zip,gzip,PNG`
- **LZ78, LZW** new block b_i is created by appending exactly one character to one of the previous b_j. `compress,GIF,TIFF,PDF`

In both cases, we make the new block as long as possible, i.e., we use greedy parsing.

Lempel-Ziv based compression schemes

Text t is partitioned into a number of disjoint blocks $b_1 b_2 \ldots b_n$. Each block is defined in terms of the blocks on the left.

What "defined" exactly means depends on the exact variant. The most common are:

- **LZ77** new block b_i is a subword of $b_1 b_2 \ldots b_{i-1}$ concatenated with exactly one character, `zip,gzip,PNG`
- **LZ78, LZW** new block b_i is created by appending exactly one character to one of the previous b_j. `compress,GIF,TIFF,PDF`

In both cases, we make the new block as long as possible, i.e., we use greedy parsing.

Lempel-Ziv based compression schemes

Text t is partitioned into a number of disjoint blocks $b_1 b_2 \ldots b_n$. Each block is defined in terms of the blocks on the left.

What "defined" exactly means depends on the exact variant. The most common are:

LZ77 new block b_i is a subword of $b_1 b_2 \ldots b_{i-1}$ concatenated with exactly one character, `zip,gzip,PNG`

LZ78, LZW new block b_i is created by appending exactly one character to one of the previous b_j.
`compress,GIF,TIFF,PDF`

In both cases, we make the new block as long as possible, i.e., we use greedy parsing.

Lempel-Ziv based compression schemes

Text t is partitioned into a number of disjoint blocks $b_1 b_2 \ldots b_n$. Each block is defined in terms of the blocks on the left.

What "defined" exactly means depends on the exact variant. The most common are:

LZ77 new block b_i is a subword of $b_1 b_2 \ldots b_{i-1}$ concatenated with exactly one character, `zip,gzip,PNG`

LZ78, LZW new block b_i is created by appending exactly one character to one of the previous b_j.
`compress,GIF,TIFF,PDF`

In both cases, we make the new block as long as possible, i.e., we use greedy parsing.

Lempel-Ziv based compression schemes

Text t is partitioned into a number of disjoint blocks $b_1 b_2 \ldots b_n$. Each block is defined in terms of the blocks on the left.

What "defined" exactly means depends on the exact variant. The most common are:

LZ77 new block b_i is a subword of $b_1 b_2 \ldots b_{i-1}$ concatenated with exactly one character, `zip,gzip,PNG`

LZ78, LZW new block b_i is created by appending exactly one character to one of the previous b_j.
`compress,GIF,TIFF,PDF`

In both cases, we make the new block as long as possible, i.e., we use greedy parsing.

Example of LZW compression:

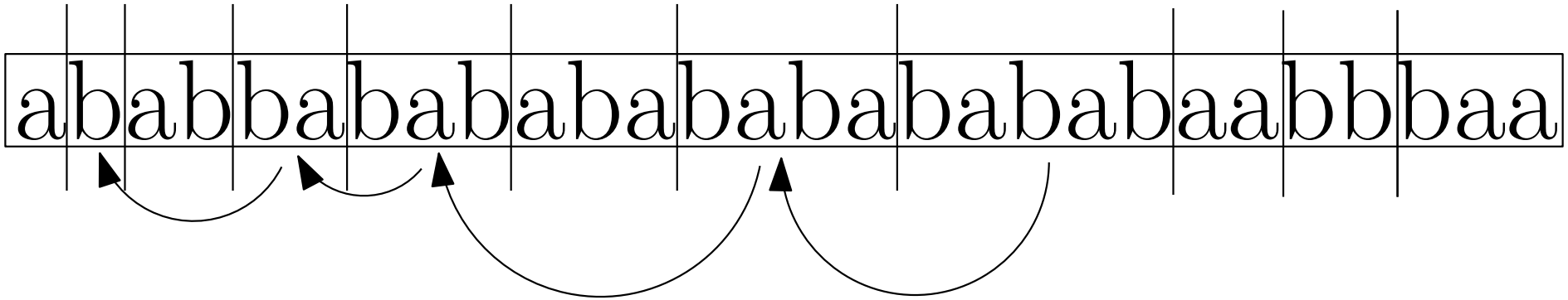
ababbababababababababbbaa

Even though $n \in \Omega(\sqrt{N})$, you can compress/decompress very quickly!

Example of LZW compression:

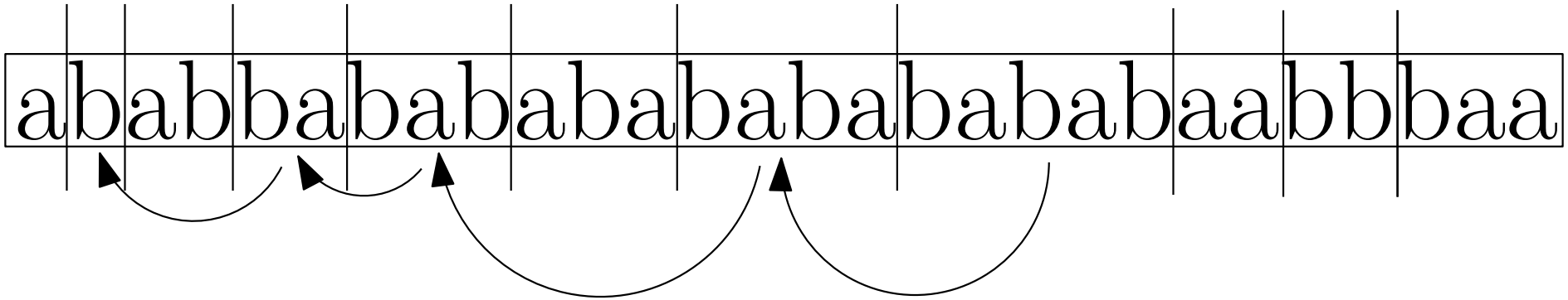
Even though $n \in \Omega(\sqrt{N})$, you can compress/decompress very quickly!

Example of LZW compression:



Even though $n \in \Omega(\sqrt{N})$, you can compress/decompress very quickly!

Example of LZW compression:



Even though $n \in \Omega(\sqrt{N})$, you can compress/decompress very quickly!

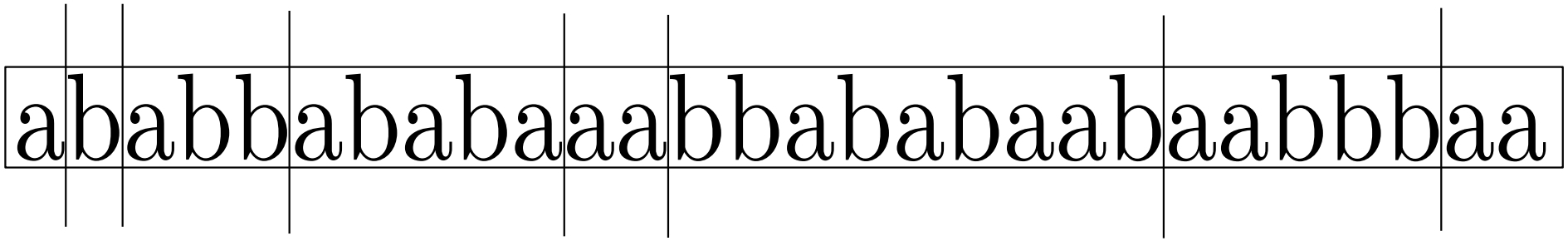
Example of LZ compression:

$$\boxed{\text{ababbababaaaabbbabababaaabaaabbbbaa}}$$

It might happen that $n = \mathcal{O}(\log N)$. Such situation is rather unlikely in practice, but it happens for Fibonacci words, and they are often considered to be **the** benchmark for string algorithms.

Sometimes we are interested in the version which is not self-referential.

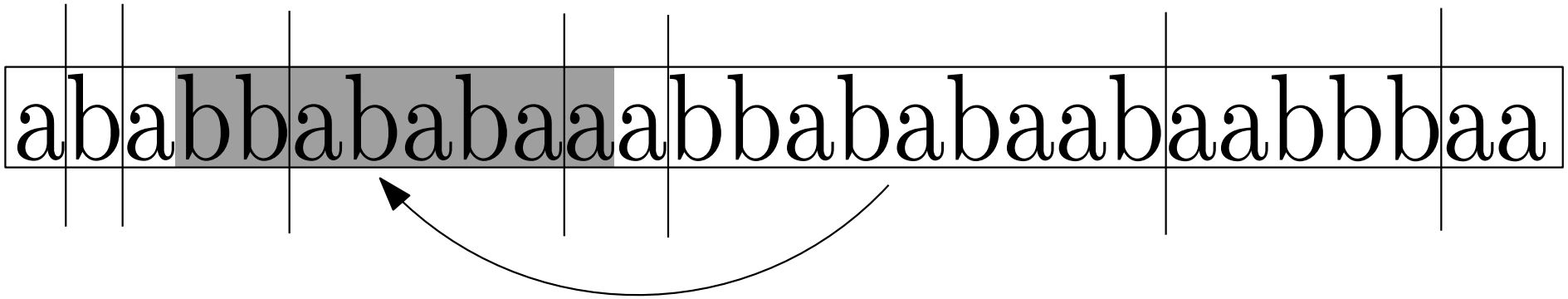
Example of LZ compression:



It might happen that $n = \mathcal{O}(\log N)$. Such situation is rather unlikely in practice, but it happens for Fibonacci words, and they are often considered to be **the** benchmark for string algorithms.

Sometimes we are interested in the version which is not self-referential.

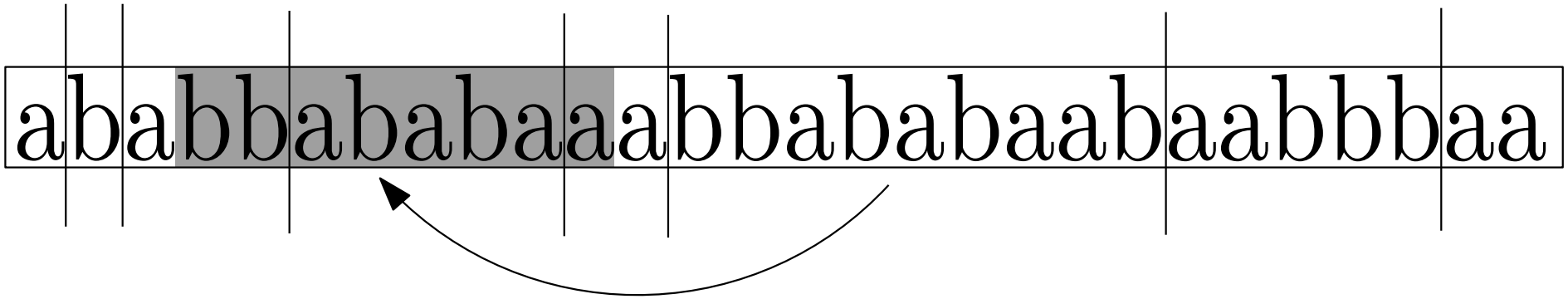
Example of LZ compression:



It might happen that $n = \mathcal{O}(\log N)$. Such situation is rather unlikely in practice, but it happens for Fibonacci words, and they are often considered to be **the** benchmark for string algorithms.

Sometimes we are interested in the version which is not self-referential.

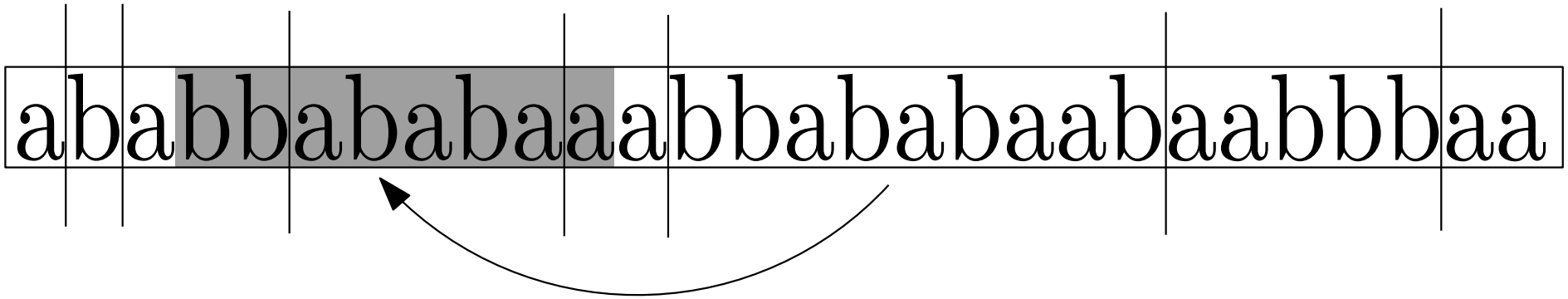
Example of LZ compression:



It might happen that $n = \mathcal{O}(\log N)$. Such situation is rather unlikely in practice, but it happens for Fibonacci words, and they are often considered to be **the** benchmark for string algorithms.

Sometimes we are interested in the version which is not self-referential.

Example of LZ compression:



It might happen that $n = \mathcal{O}(\log N)$. Such situation is rather unlikely in practice, but it happens for Fibonacci words, and they are often considered to be **the** benchmark for string algorithms.

Sometimes we are interested in the version which is not self-referential.

We will solve a more difficult problem.

Longest previous substring

For a given string $w[1..n]$ we define an array $\mathrm{LPF}[1..n]$. $\mathrm{LPF}[i]$ is the largest k such that $w[i..i+k-1]$ occurs starting somewhere on the left of i, formally $w[i..i+k-1] = w[i'..i'+k-1]$ for some $i' < i$.

Note that the previous occurrence is allowed to overlap with $w[i..i+k-1]$!

We will solve a more difficult problem.

Longest previous substring

For a given string $w[1..n]$ we define an array $\mathrm{LPF}[1..n]$. $\mathrm{LPF}[i]$ is the largest k such that $w[i..i+k-1]$ occurs starting somewhere on the left of i, formally $w[i..i+k-1] = w[i'..i'+k-1]$ for some $i' < i$.

Note that the previous occurrence is allowed to overlap with $w[i..i+k-1]$!

We will solve a more difficult problem.

Longest previous substring

For a given string $w[1..n]$ we define an array $\mathrm{LPF}[1..n]$. $\mathrm{LPF}[i]$ is the largest k such that $w[i..i+k-1]$ occurs starting somewhere on the left of i, formally $w[i..i+k-1] = w[i'..i'+k-1]$ for some $i' < i$.

Note that the previous occurrence is allowed to overlap with $w[i..i+k-1]$!

We will solve a more difficult problem.

Longest previous substring

For a given string $w[1..n]$ we define an array $\mathrm{LPF}[1..n]$. $\mathrm{LPF}[i]$ is the largest k such that $w[i..i+k-1]$ occurs starting somewhere on the left of i, formally $w[i..i+k-1] = w[i'..i'+k-1]$ for some $i' < i$.

Note that the previous occurrence is allowed to overlap with $w[i..i+k-1]$!

Having the $\mathrm{LPF}[1..n]$ table, we can generate the self-referential LZ parse as follows.

- Start with $i = 1$.
- Output $w[i..i + \mathrm{LPF}[i]]$ as the next block.
- Increase i by $\mathrm{LPF}[i] + 1$.
- Repeat.

Message to take back home

Sometimes it pays off to reduce a more complicated problem.

Having the $\mathrm{LPF}[1..n]$ table, we can generate the self-referential LZ parse as follows.

- Start with $i = 1$.
- Output $w[i..i + \mathrm{LPF}[i]]$ as the next block.
- Increase i by $\mathrm{LPF}[i] + 1$.
- Repeat.

Message to take back home

Sometimes it pays off to reduce a more complicated problem.

Having the $\mathrm{LPF}[1..n]$ table, we can generate the self-referential LZ parse as follows.

- Start with $i = 1$.
- Output $w[i..i + \mathrm{LPF}[i]]$ as the next block.
- Increase i by $\mathrm{LPF}[i] + 1$.
- Repeat.

Message to take back home

Sometimes it pays off to reduce a more complicated problem.

Having the $\mathrm{LPF}[1..n]$ table, we can generate the self-referential LZ parse as follows.

- Start with $i = 1$.
- Output $w[i..i + \mathrm{LPF}[i]]$ as the next block.
- Increase i by $\mathrm{LPF}[i] + 1$.
- Repeat.

Message to take back home

Sometimes it pays off to reduce a more complicated problem.

Having the $\mathrm{LPF}[1..n]$ table, we can generate the self-referential LZ parse as follows.

- Start with $i = 1$.
- Output $w[i..i + \mathrm{LPF}[i]]$ as the next block.
- Increase i by $\mathrm{LPF}[i] + 1$.
- Repeat.

Message to take back home

Sometimes it pays off to reduce a more complicated problem.

Having the $\mathrm{LPF}[1..n]$ table, we can generate the self-referential LZ parse as follows.

- Start with $i = 1$.
- Output $w[i..i + \mathrm{LPF}[i]]$ as the next block.
- Increase i by $\mathrm{LPF}[i] + 1$.
- Repeat.

Message to take back home

Sometimes it pays off to reduce a more complicated problem.

Now we focus on computing all $\mathrm{LPF}[i]$.

For each i we want to maximize over all $i' < i$ the longest common prefix of $w[i..n]$ and $w[i'..n]$.

This is difficult, so let's try something simpler.

For each i we want to maximize over all $i' \neq i$ the longest common prefix of $w[i..n]$ and $w[i'..n]$.

This is simple! Just take the predecessor/successor of i in the suffix array. Check which one is better by either executing two LCP queries (we know how to do implement them in constant time), or just use the lcp array (much simpler).

Now we focus on computing all $\mathrm{LPF}[i]$.

For each i we want to maximize over all $i' < i$ the longest common prefix of $w[i..n]$ and $w[i'..n]$.

This is difficult, so let's try something simpler.

For each i we want to maximize over all $i' \neq i$ the longest common prefix of $w[i..n]$ and $w[i'..n]$.

This is simple! Just take the predecessor/successor of i in the suffix array. Check which one is better by either executing two LCP queries (we know how to do implement them in constant time), or just use the lcp array (much simpler).

Now we focus on computing all $\mathrm{LPF}[i]$.

For each i we want to maximize over all $i' < i$ the longest common prefix of $w[i..n]$ and $w[i'..n]$.

This is difficult, so let's try something simpler.

For each i we want to maximize over all $i' \neq i$ the longest common prefix of $w[i..n]$ and $w[i'..n]$.

This is simple! Just take the predecessor/successor of i in the suffix array. Check which one is better by either executing two LCP queries (we know how to do implement them in constant time), or just use the lcp array (much simpler).

Now, it might happen that we will get $i' > i$, which is not really useful. But for $i = n$ this cannot happen, hence we know how to compute $\mathrm{LPF}[n]$ in constant time. What then?

Remove $w[n..n]$ from the suffix array. Then by looking at the predecessor/successor of $n-1$ in the (remaining part of the) suffix array we get $i' \neq n-1$ maximizing the longest common prefix, which is the same as $i' < n-1$ maximizing the longest common prefix, so we can compute $\mathrm{LPF}[n-1]$.

Remove $w[n-1..n]$ from the suffix array, and look at the predecessor/successor of $n-2$ in the (remaining part of the) suffix array to compute $\mathrm{LPF}[n-2]$.

Now, it might happen that we will get $i' > i$, which is not really useful. But for $i = n$ this cannot happen, hence we know how to compute $\mathrm{LPF}[n]$ in constant time. What then?

Remove $w[n..n]$ from the suffix array. Then by looking at the predecessor/successor of $n-1$ in the (remaining part of the) suffix array we get $i' \neq n-1$ maximizing the longest common prefix, which is the same as $i' < n-1$ maximizing the longest common prefix, so we can compute $\mathrm{LPF}[n-1]$.

Remove $w[n-1..n]$ from the suffix array, and look at the predecessor/successor of $n-2$ in the (remaining part of the) suffix array to compute $\mathrm{LPF}[n-2]$.

Now, it might happen that we will get $i' > i$, which is not really useful. But for $i = n$ this cannot happen, hence we know how to compute $\mathrm{LPF}[n]$ in constant time. What then?

Remove $w[n..n]$ from the suffix array. Then by looking at the predecessor/successor of $n-1$ in the (remaining part of the) suffix array we get $i' \neq n-1$ maximizing the longest common prefix, which is the same as $i' < n-1$ maximizing the longest common prefix, so we can compute $\mathrm{LPF}[n-1]$.

Remove $w[n-1..n]$ from the suffix array, and look at the predecessor/successor of $n-2$ in the (remaining part of the) suffix array to compute $\mathrm{LPF}[n-2]$.

This works in constant time per each i assuming that we can:

- maintain the suffix array under removing any $w[i..n]$ and retrieving the predecessor/successor of any $w[i..n]$,
- compute the longest common prefix of any $w[i..n]$ and its predecessor/successor in the remaining part of the suffix array.

For the first part, just use a doubly-linked list to store SA. For the second part, the lazy way is to apply the whole lcp machinery, i.e., assume that we can compute the longest common prefix of any two suffixes in constant time.

This works in constant time per each i assuming that we can:

- maintain the suffix array under removing any $w[i..n]$ and retrieving the predecessor/successor of any $w[i..n]$,
- compute the longest common prefix of any $w[i..n]$ and its predecessor/successor in the remaining part of the suffix array.

For the first part, just use a doubly-linked list to store SA. For the second part, the lazy way is to apply the whole lcp machinery, i.e., assume that we can compute the longest common prefix of any two suffixes in constant time.

This works in constant time per each i assuming that we can:

- maintain the suffix array under removing any $w[i..n]$ and retrieving the predecessor/successor of any $w[i..n]$,
- compute the longest common prefix of any $w[i..n]$ and its predecessor/successor in the remaining part of the suffix array.

For the first part, just use a doubly-linked list to store SA. For the second part, the lazy way is to apply the whole lcp machinery, i.e., assume that we can compute the longest common prefix of any two suffixes in constant time.

This works in constant time per each i assuming that we can:

- maintain the suffix array under removing any $w[i..n]$ and retrieving the predecessor/successor of any $w[i..n]$,
- compute the longest common prefix of any $w[i..n]$ and its predecessor/successor in the remaining part of the suffix array.

For the first part, just use a doubly-linked list to store SA. For the second part, the lazy way is to apply the whole lcp machinery, i.e., assume that we can compute the longest common prefix of any two suffixes in constant time.

Simpler solution for the second part: maintain not only SA, but also lcp, which are the the longest common prefixes between any two neighbors in the current suffix array.

Say that we remove an element from the current suffix array. So, we had

$$..., w[x..n], w[y..n], w[z..n], ...$$

and we want to leave

$$..., w[x..n], w[z..n], ...$$

We knew $\mathrm{lcp}(w[x..n], w[y..n])$ and $\mathrm{lcp}(w[y..n], w[z..n])$, Then, $\mathrm{lcp}(w[x..n], w[z..n])$ is simply the minimum of these two, so we can maintain all data in constant time per update.

All $\mathrm{LPF}[i]$ can be computed in constant time per entry.

Simpler solution for the second part: maintain not only SA, but also lcp, which are the the longest common prefixes between any two neighbors in the current suffix array.
Say that we remove an element from the current suffix array. So, we had

$$\ldots, w[x..n], w[y..n], w[z..n], \ldots$$

and we want to leave

$$\ldots, w[x..n], w[z..n], \ldots$$

We knew $\mathrm{lcp}(w[x..n], w[y..n])$ and $\mathrm{lcp}(w[y..n], w[z..n])$, Then, $\mathrm{lcp}(w[x..n], w[z..n])$ is simply the minimum of these two, so we can maintain all data in constant time per update.

All $\mathrm{LPF}[i]$ can be computed in constant time per entry.

Simpler solution for the second part: maintain not only SA, but also lcp, which are the the longest common prefixes between any two neighbors in the current suffix array.
Say that we remove an element from the current suffix array. So, we had

$$\ldots, w[x..n], w[y..n], w[z..n], \ldots$$

and we want to leave

$$\ldots, w[x..n], w[z..n], \ldots$$

We knew $\mathrm{lcp}(w[x..n], w[y..n])$ and $\mathrm{lcp}(w[y..n], w[z..n])$, Then, $\mathrm{lcp}(w[x..n], w[z..n])$ is simply the minimum of these two, so we can maintain all data in constant time per update.

All $\mathrm{LPF}[i]$ can be computed in constant time per entry.

Simpler solution for the second part: maintain not only SA, but also lcp, which are the the longest common prefixes between any two neighbors in the current suffix array.
Say that we remove an element from the current suffix array. So, we had

$$\ldots, w[x..n], w[y..n], w[z..n], \ldots$$

and we want to leave

$$\ldots, w[x..n], w[z..n], \ldots$$

We knew $\mathrm{lcp}(w[x..n], w[y..n])$ and $\mathrm{lcp}(w[y..n], w[z..n])$, Then, $\mathrm{lcp}(w[x..n], w[z..n])$ is simply the minimum of these two, so we can maintain all data in constant time per update.

All $\mathrm{LPF}[i]$ can be computed in constant time per entry.

Simpler solution for the second part: maintain not only SA, but also lcp, which are the the longest common prefixes between any two neighbors in the current suffix array.
Say that we remove an element from the current suffix array. So, we had

$$\ldots, w[x..n], w[y..n], w[z..n], \ldots$$

and we want to leave

$$\ldots, w[x..n], w[z..n], \ldots$$

We knew $\mathrm{lcp}(w[x..n], w[y..n])$ and $\mathrm{lcp}(w[y..n], w[z..n])$, Then, $\mathrm{lcp}(w[x..n], w[z..n])$ is simply the minimum of these two, so we can maintain all data in constant time per update.

All $\mathrm{LPF}[i]$ can be computed in constant time per entry.

Simpler solution for the second part: maintain not only SA, but also lcp, which are the the longest common prefixes between any two neighbors in the current suffix array.
Say that we remove an element from the current suffix array. So, we had

$$\ldots, w[x..n], w[y..n], w[z..n], \ldots$$

and we want to leave

$$\ldots, w[x..n], w[z..n], \ldots$$

We knew $\mathrm{lcp}(w[x..n], w[y..n])$ and $\mathrm{lcp}(w[y..n], w[z..n])$, Then, $\mathrm{lcp}(w[x..n], w[z..n])$ is simply the minimum of these two, so we can maintain all data in constant time per update.

All $\mathrm{LPF}[i]$ can be computed in constant time per entry.

Now we move to another compression method: Burrows-Wheeler transform. Technically speaking, it is not a compression method by itself: given a string $w[1..n]$, it produces a (probably different) string of length $n + 1$.

For `mississippi`, the transform returns `ipssm$pissii`. Hmm.

So why do we care? The transformed string will be easier to compress with some simple tools. So, we first apply the transform, then use some simple compression scheme. Hopefully, the result will be better than applying the simple compression scheme directly on the original string.

The transform needs to be efficiently **reversible**.

Now we move to another compression method: Burrows-Wheeler transform. Technically speaking, it is not a compression method by itself: given a string $w[1..n]$, it produces a (probably different) string of length $n + 1$.

For `mississippi`, the transform returns `ipssm$pissii`. Hmm.

So why do we care? The transformed string will be easier to compress with some simple tools. So, we first apply the transform, then use some simple compression scheme. Hopefully, the result will be better than applying the simple compression scheme directly on the original string.

The transform needs to be efficiently **reversible**.

Now we move to another compression method: Burrows-Wheeler transform. Technically speaking, it is not a compression method by itself: given a string $w[1..n]$, it produces a (probably different) string of length $n + 1$.

For `mississippi`, the transform returns `ipssm$pissii`. Hmm.

So why do we care? The transformed string will be easier to compress with some simple tools. So, we first apply the transform, then use some simple compression scheme. Hopefully, the result will be better than applying the simple compression scheme directly on the original string.

The transform needs to be efficiently **reversible**.

Now we move to another compression method: Burrows-Wheeler transform. Technically speaking, it is not a compression method by itself: given a string $w[1..n]$, it produces a (probably different) string of length $n + 1$.

For `mississippi`, the transform returns `ipssm$pissii`. Hmm.

So why do we care? The transformed string will be easier to compress with some simple tools. So, we first apply the transform, then use some simple compression scheme. Hopefully, the result will be better than applying the simple compression scheme directly on the original string.

The transform needs to be efficiently **reversible**.

Take your string, append a special character \$, and construct all cyclic shifts of the resulting word of length $|w| + 1$.

$w =$ mississippi\$

mississippi\$	\$mississippi	\$
ississippi\$m	i\$mississipp	i\$
ssissippi\$mi	ippi\$mississ	ippi\$
sissippi\$mis	issippi\$miss	issippi\$
issippi\$miss	ississippi\$m	ississippi\$
ssippi\$missi	mississippi\$	mississippi\$
sippi\$missis	pi\$mississip	pi\$
ippi\$mississ	ppi\$mississi	ppi\$
ppi\$mississi	sippi\$missis	sippi\$
pi\$mississip	sissippi\$mis	sissippi\$
i\$mississipp	ssippi\$missi	ssippi\$
\$mississippi	ssissippi\$mi	ssissippi\$

Take your string, append a special character \$, and construct all cyclic shifts of the resulting word of length $|w| + 1$.

$w =$ mississippi\$

mississippi\$	\$mississippi	\$
ississippi\$m	i\$mississipp	i\$
ssissippi\$mi	ippi\$mississ	ippi\$
sissippi\$mis	issippi\$miss	issippi\$
issippi\$miss	ississippi\$m	ississippi\$
ssippi\$missi	mississippi\$	mississippi\$
sippi\$missis	pi\$mississip	pi\$
ippi\$mississ	ppi\$mississi	ppi\$
ppi\$mississi	sippi\$missis	sippi\$
pi\$mississip	sissippi\$mis	sissippi\$
i\$mississipp	ssippi\$missi	ssippi\$
\$mississippi	ssissippi\$mi	ssissippi\$

The last column of the sorted array is the Burrows-Wheeler transformed text.

w = mississippi$

mississippi$	$mississippi	$
ississippi$m	i$mississipp	i$
ssissippi$mi	ippi$mississ	ippi$
sissippi$mis	issippi$miss	issippi$
issippi$miss	ississippi$m	ississippi$
ssippi$missi	mississippi$	mississippi$
sippi$missis	pi$mississip	pi$
ippi$mississ	ppi$mississi	ppi$
ppi$mississi	sippi$missis	sippi$
pi$mississip	sissippi$mis	sissippi$
i$mississipp	ssippi$missi	ssippi$
$mississippi	ssissippi$mi	ssissippi$

The last column of the sorted array is the Burrows-Wheeler transformed text.

w = mississippi$

mississippi$	$mississippi	$
ississippi$m	i$mississipp	i$
ssissippi$mi	ippi$mississ	ippi$
sissippi$mis	issippi$miss	issippi$
issippi$miss	ississippi$m	ississippi$
ssippi$missi	mississippi$	mississippi$
sippi$mississ	pi$mississip	pi$
ippi$mississ	ppi$mississi	ppi$
ppi$mississi	sippi$missis	sippi$
pi$mississip	sissippi$mis	sissippi$
i$mississipp	ssippi$missi	ssippi$
$mississippi	ssissippi$mi	ssissippi$

How to compute the transform?

Lemma

BWT can be computed in linear time by constructing the suffix array of $w\$$.

So, that's easy. But can we reverse the transform efficiently, too?

First column of the sorted array is called F, and the last column is called L. So, BWT is really L.

We define the L-to-F mapping.

$\mathrm{LF}[i] = j$

We shift the word in the i-th row of the sorted array by one to the right and locate the j-th row of the sorted array containing the result.

Take the first row, which contain `$mississippi`. Shift it by one to get `i$mississipp`. The result is in the second row, so $\mathrm{LF}[1] = 2$.

First column of the sorted array is called F, and the last column is called L. So, BWT is really L.

We define the L-to-F mapping.

$\mathrm{LF}[i] = j$

We shift the word in the i-th row of the sorted array by one to the right and locate the j-th row of the sorted array containing the result.

Take the first row, which contain `$mississippi`. Shift it by one to get `i$mississipp`. The result is in the second row, so $\mathrm{LF}[1] = 2$.

First column of the sorted array is called F, and the last column is called L. So, BWT is really L.

We define the L-to-F mapping.

$\mathrm{LF}[i] = j$

We shift the word in the i-th row of the sorted array by one to the right and locate the j-th row of the sorted array containing the result.

Take the first row, which contain `$mississippi`. Shift it by one to get `i$mississipp`. The result is in the second row, so $\mathrm{LF}[1] = 2$.

First column of the sorted array is called F, and the last column is called L. So, BWT is really L.

We define the L-to-F mapping.

$\mathrm{LF}[i] = j$

We shift the word in the i-th row of the sorted array by one to the right and locate the j-th row of the sorted array containing the result.

Take the first row, which contain `$mississippi`. Shift it by one to get `i$mississipp`. The result is in the second row, so $\mathrm{LF}[1] = 2$.

$w =$ mississippi$

		i	LF[i]
mississippi$	$mississippi	1	2
ississippi$m	i$mississipp	2	7
ssissippi$mi	ippi$mississ	3	9
sissippi$mis	issippi$miss	4	10
issippi$miss	ississippi$m	5	6
ssippi$missi	mississippi$	6	1
sippi$mississ	pi$mississip	7	8
ippi$mississ	ppi$mississi	8	3
ppi$mississi	sippi$missis	9	11
pi$mississip	sissippi$mis	10	12
i$mississipp	ssippi$missi	11	4
$mississippi	ssissippi$mi	12	5

Now look at L[1], L[LF[1]], L[LF[LF[1]]],...

$w =$ mississippi$

		i	LF[i]
mississippi$	$mississippi	1	2
ississippi$m	i$mississipp	2	7
ssissippi$mi	ippi$mississ	3	9
sissippi$mis	issippi$miss	4	10
issippi$miss	ississippi$m	5	6
ssippi$missi	mississippi$	6	1
sippi$mississ	pi$mississip	7	8
ippi$mississ	ppi$mississi	8	3
ppi$mississi	sippi$missis	9	11
pi$mississip	sissippi$mis	10	12
i$mississipp	ssippi$missi	11	4
$mississippi	ssissippi$mi	12	5

Now look at L[1], L[LF[1]], L[LF[LF[1]]],...

$w = \texttt{mississippi\$}$

		i	LF[i]
mississippi$	$mississippi	1	2
ississippi$m	i$mississipp	2	7
ssissippi$mi	ippi$mississ	3	9
sissippi$mis	issippi$miss	4	10
issippi$miss	ississippi$m	5	6
ssippi$missi	mississippi$	6	1
sippi$mississ	pi$mississip	7	8
ippi$mississ	ppi$mississi	8	3
ppi$mississi	sippi$mississ	9	11
pi$mississip	sissippi$mis	10	12
i$mississipp	ssippi$missi	11	4
$mississippi	ssissippi$mi	12	5

Now look at L[1], L[LF[1]], L[LF[LF[1]]],...

$w =$ mississippi$

		i	LF[i]
mississippi$	$mississippi	1	2
ississippi$m	i$mississipp	2	7
ssissippi$mi	ippi$mississ	3	9
sissippi$mis	issippi$miss	4	10
issippi$miss	ississippi$m	5	6
ssippi$missi	mississippi$	6	1
sippi$mississ	pi$mississip	7	8
ippi$mississ	ppi$mississi	8	3
ppi$mississi	sippi$mississ	9	11
pi$mississip	sissippi$mis	10	12
i$mississipp	ssippi$missi	11	4
$mississippi	ssissippi$mi	12	5

Now look at L[1], L[LF[1]], L[LF[LF[1]]],...

Observation

$$w[|w| - i] = \mathrm{L}[\mathrm{LF}^i[1]]$$

So, having the array LF allows us to actually reconstruct w. But where do we get this array from?

- $C[x]$ is the the total number of all characters $c < x$ in the text.
- $Occ(x, i)$ is the number of occurrences of the letter x in $\mathrm{L}[1..i]$.

Lemma

$$\mathrm{LF}[i] = \mathrm{C}[\mathrm{L}[i]] + Occ(\mathrm{L}[i], i)$$

Observation

$$w[|w| - i] = \mathrm{L}[\mathrm{LF}^i[1]]$$

So, having the array LF allows us to actually reconstruct w. But where do we get this array from?

- $C[x]$ is the the total number of all characters $c < x$ in the text.
- $Occ(x, i)$ is the number of occurrences of the letter x in $\mathrm{L}[1..i]$.

Lemma

$$\mathrm{LF}[i] = \mathrm{C}[\mathrm{L}[i]] + Occ(\mathrm{L}[i], i)$$

Observation

$$w[|w| - i] = \mathrm{L}[\mathrm{LF}^i[1]]$$

So, having the array LF allows us to actually reconstruct w. But where do we get this array from?

- $C[x]$ is the the total number of all characters $c < x$ in the text.
- $Occ(x, i)$ is the number of occurrences of the letter x in $\mathrm{L}[1..i]$.

Lemma

$$\mathrm{LF}[i] = \mathrm{C}[\mathrm{L}[i]] + Occ(\mathrm{L}[i], i)$$

Observation

$$w[|w| - i] = \mathrm{L}[\mathrm{LF}^i[1]]$$

So, having the array LF allows us to actually reconstruct w. But where do we get this array from?

- $C[x]$ is the the total number of all characters $c < x$ in the text.
- $Occ(x, i)$ is the number of occurrences of the letter x in $\mathrm{L}[1..i]$.

Lemma

$$\mathrm{LF}[i] = \mathrm{C}[\mathrm{L}[i]] + Occ(\mathrm{L}[i], i)$$

Observation

$$w[|w| - i] = \mathrm{L}[\mathrm{LF}^i[1]]$$

So, having the array LF allows us to actually reconstruct w. But where do we get this array from?

- $C[x]$ is the the total number of all characters $c < x$ in the text.
- $Occ(x, i)$ is the number of occurrences of the letter x in $\mathrm{L}[1..i]$.

Lemma

$$\mathrm{LF}[i] = \mathrm{C}[\mathrm{L}[i]] + Occ(\mathrm{L}[i], i)$$

This actually gives a linear time reconstruction algorithm if we can implement $Occ(x, i)$ efficiently. For that, observe that we only need $Occ(\mathrm{L}[i], i)$, which can be computed in a single left-to-right scan over L. More precisely, we scan and maintain a table counting the occurrences of each letter seen so far.

Theorem

BWT and its reverse can be both computed in linear time.

This actually gives a linear time reconstruction algorithm if we can implement $Occ(x, i)$ efficiently. For that, observe that we only need $Occ(\mathrm{L}[i], i)$, which can be computed in a single left-to-right scan over L. More precisely, we scan and maintain a table counting the occurrences of each letter seen so far.

Theorem

BWT and its reverse can be both computed in linear time.

This actually gives a linear time reconstruction algorithm if we can implement $Occ(x, i)$ efficiently. For that, observe that we only need $Occ(\mathrm{L}[i], i)$, which can be computed in a single left-to-right scan over L. More precisely, we scan and maintain a table counting the occurrences of each letter seen so far.

Theorem

BWT and its reverse can be both computed in linear time.

This actually gives a linear time reconstruction algorithm if we can implement $Occ(x, i)$ efficiently. For that, observe that we only need $Occ(\mathrm{L}[i], i)$, which can be computed in a single left-to-right scan over L. More precisely, we scan and maintain a table counting the occurrences of each letter seen so far.

Theorem

BWT and its reverse can be both computed in linear time.

Burrows-Wheeler transform can also be used for pattern matching. This is very useful when space is an issue, because the transformed text can be stored in a compressed form.

We will very briefly discuss another structure for text indexing: suffix trees.

Burrows-Wheeler transform can also be used for pattern matching. This is very useful when space is an issue, because the transformed text can be stored in a compressed form.

We will very briefly discuss another structure for text indexing: suffix trees.

Suffix tree $ST(w[1..n])$

We append a special terminating character \$ to our word $w[1..n]$. Then we arrange all suffixes of $w[1..n]$\$ in a compacted trie.

Take a *banana*. The suffixes are *a*\$, *na*\$, *ana*\$, *nana*\$, *anana*\$, *banana*\$.

Suffix tree $ST(w[1..n])$

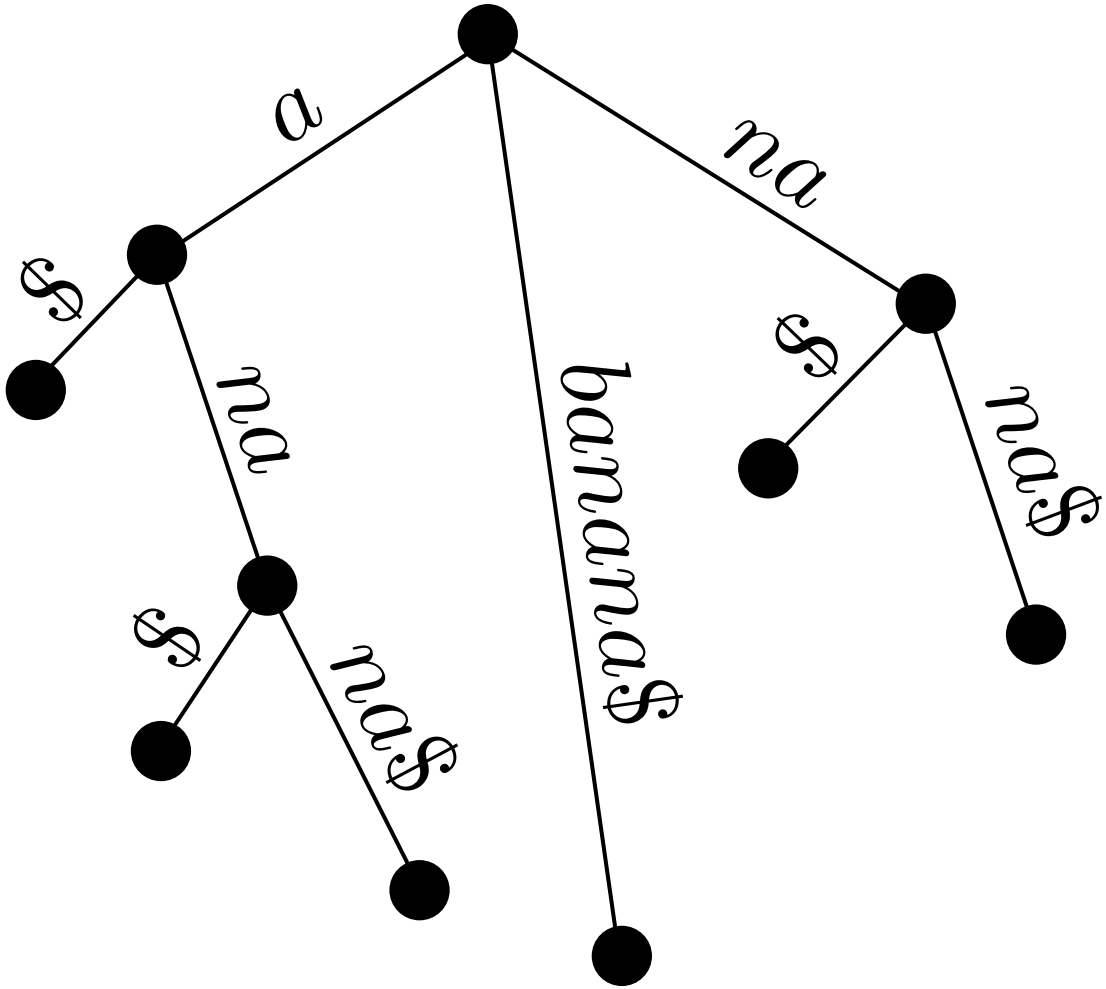
We append a special terminating character \$ to our word $w[1..n]$. Then we arrange all suffixes of $w[1..n]$\$ in a compacted trie.

Take a *banana*. The suffixes are *a*\$, *na*\$, *ana*\$, *nana*\$, *anana*\$, *banana*\$.

Suffix tree $ST(w[1..n])$

We append a special terminating character \$ to our word $w[1..n]$. Then we arrange all suffixes of $w[1..n]\$$ in a compacted trie.

Take a *banana*. The suffixes are *a*\$, *na*\$, *ana*\$, *nana*\$, *anana*\$, *banana*\$.

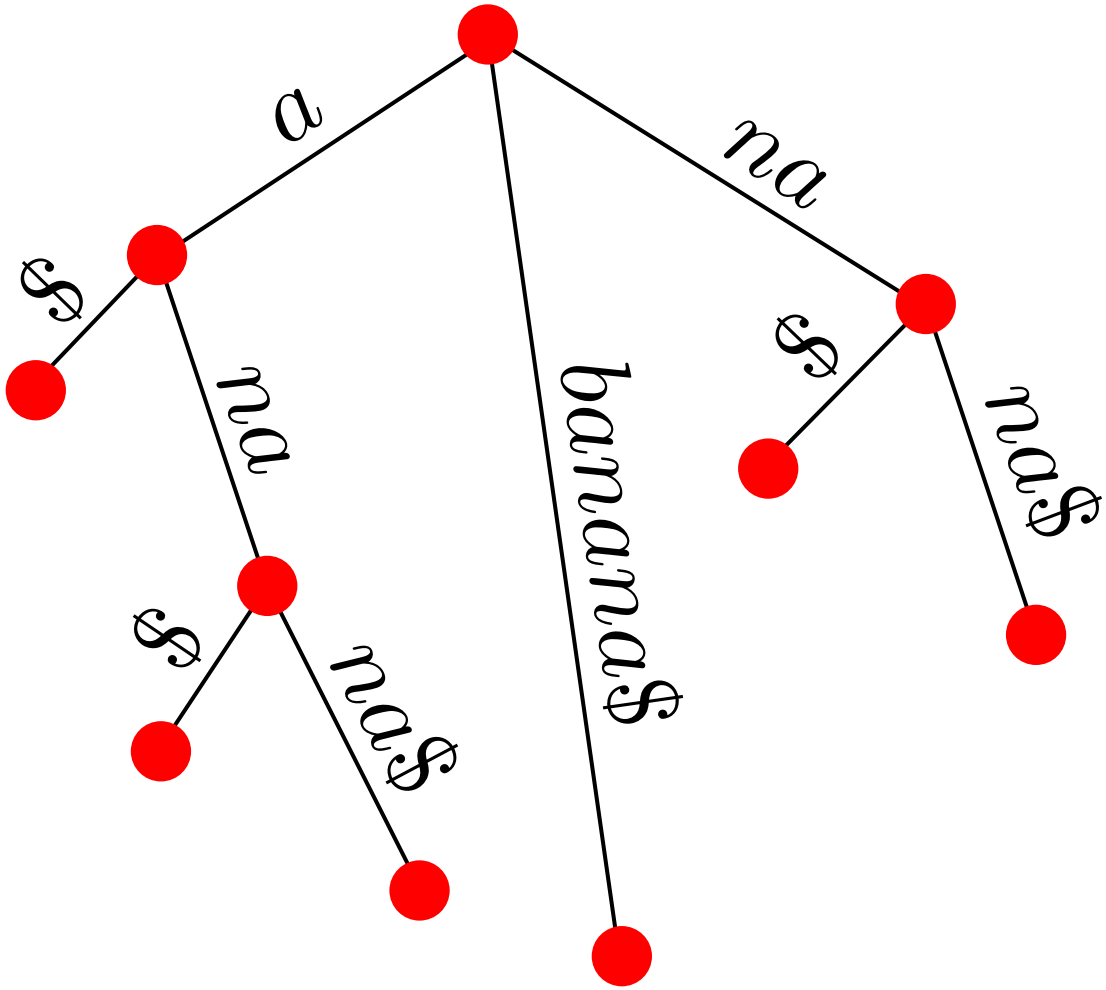


Why?

The resulting structure represents all subwords of $w[1..n]$. Each such subword is an explicit or implicit node of the suffix tree.

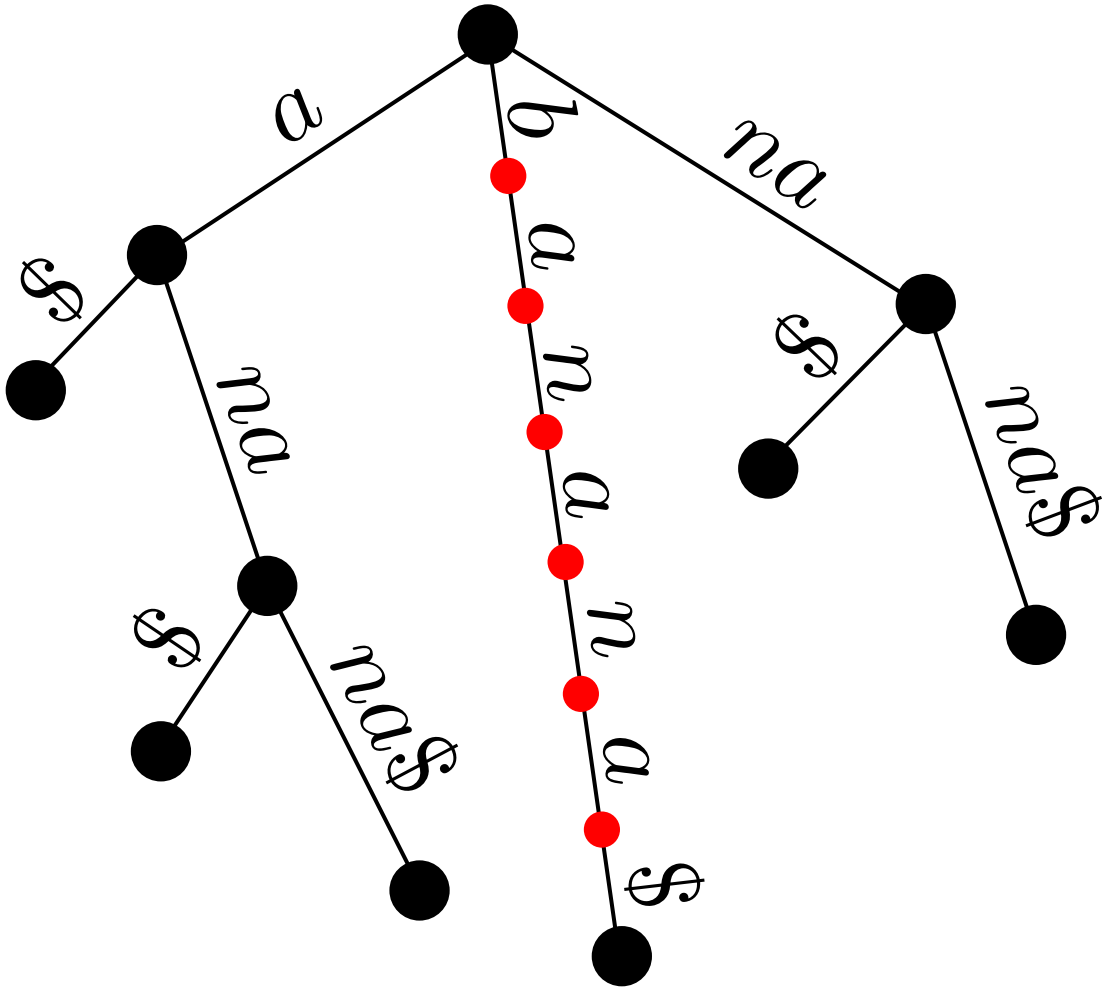
Why?

The resulting structure represents all subwords of $w[1..n]$. Each such subword is an explicit or implicit node of the suffix tree.



Why?

The resulting structure represents all subwords of $w[1..n]$. Each such subword is an explicit or implicit node of the suffix tree.



So, a suffix tree allows us to index the the input word. We keep only the explicit nodes, and there are n of them. The labels of the edges are not kept explicitly: we just remember where they occur in $w[1..n]$.

The total size of the structure is $\mathcal{O}(n)$ and all occ occurrences of a pattern $p[1..m]$ can be generated in $\mathcal{O}(m + occ)$ time.

It is not very difficult to construct a suffix tree from a suffix array in $\mathcal{O}(n)$ time. Do you see any advantage of suffix tree over suffix array? Or vice versa?

So, a suffix tree allows us to index the the input word. We keep only the explicit nodes, and there are n of them. The labels of the edges are not kept explicitly: we just remember where they occur in $w[1..n]$.

The total size of the structure is $\mathcal{O}(n)$ and all occ occurrences of a pattern $p[1..m]$ can be generated in $\mathcal{O}(m + occ)$ time.

It is not very difficult to construct a suffix tree from a suffix array in $\mathcal{O}(n)$ time. Do you see any advantage of suffix tree over suffix array? Or vice versa?

So, a suffix tree allows us to index the the input word. We keep only the explicit nodes, and there are n of them. The labels of the edges are not kept explicitly: we just remember where they occur in $w[1..n]$.

The total size of the structure is $\mathcal{O}(n)$ and all occ occurrences of a pattern $p[1..m]$ can be generated in $\mathcal{O}(m + occ)$ time.

It is not very difficult to construct a suffix tree from a suffix array in $\mathcal{O}(n)$ time. Do you see any advantage of suffix tree over suffix array? Or vice versa?

So, a suffix tree allows us to index the the input word. We keep only the explicit nodes, and there are n of them. The labels of the edges are not kept explicitly: we just remember where they occur in $w[1..n]$.

The total size of the structure is $\mathcal{O}(n)$ and all occ occurrences of a pattern $p[1..m]$ can be generated in $\mathcal{O}(m + occ)$ time.

It is not very difficult to construct a suffix tree from a suffix array in $\mathcal{O}(n)$ time. Do you see any advantage of suffix tree over suffix array? Or vice versa?

Questions?