max planck institut
informatik

Lempel-Ziv compression: how and why?

Algorithms on Strings

Paweł Gawrychowski

July 9, 2013

S

Outline

Today and next week we are going to talk about compression. 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 till the next week!

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`

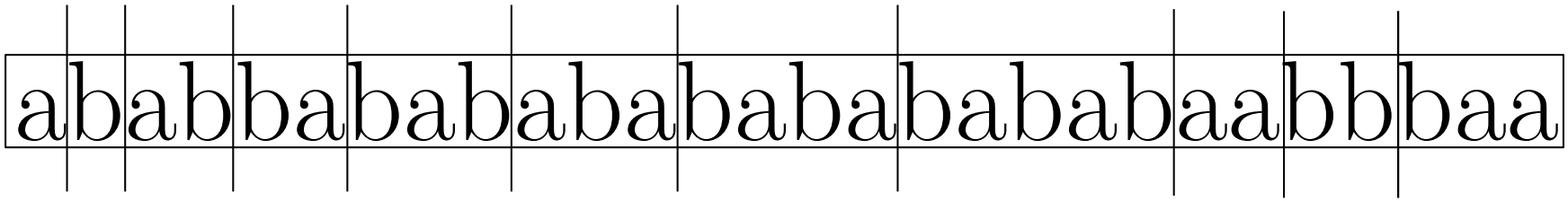
In both cases, we make the new block as long as possible, i.e., we use greedy parsing.

Example of LZW compression:

ababbababababababababaabbbaa

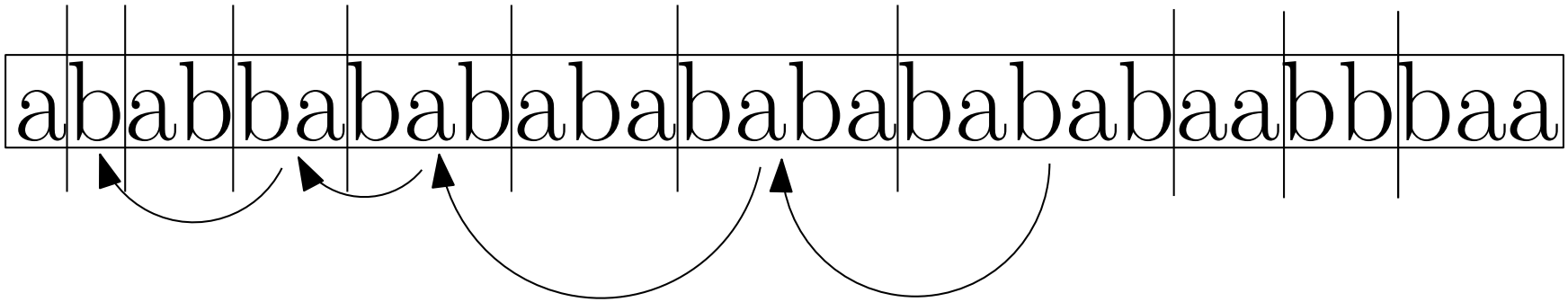
Even though $n \in \Omega(\sqrt{N})$, you can compress/decompress very quickly!

Example of LZW compression:



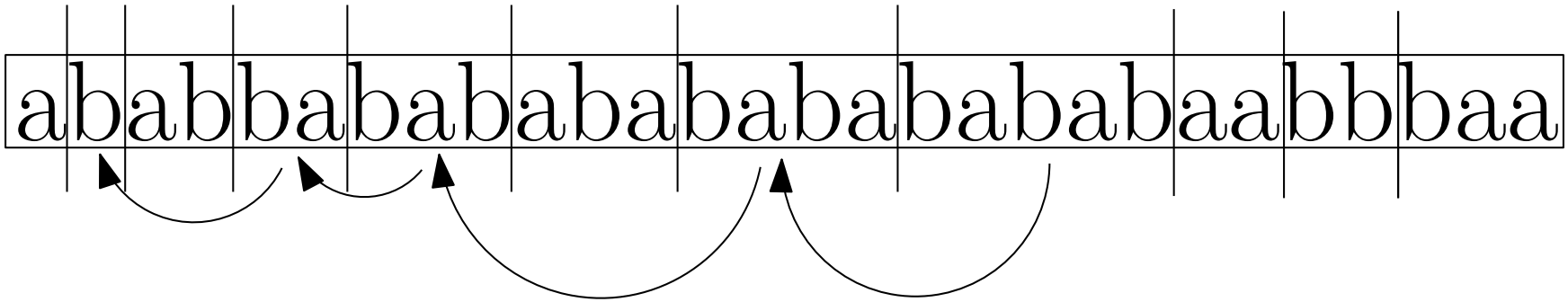
Even though $n \in \Omega(\sqrt{N})$, you can compress/decompress very quickly!

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!

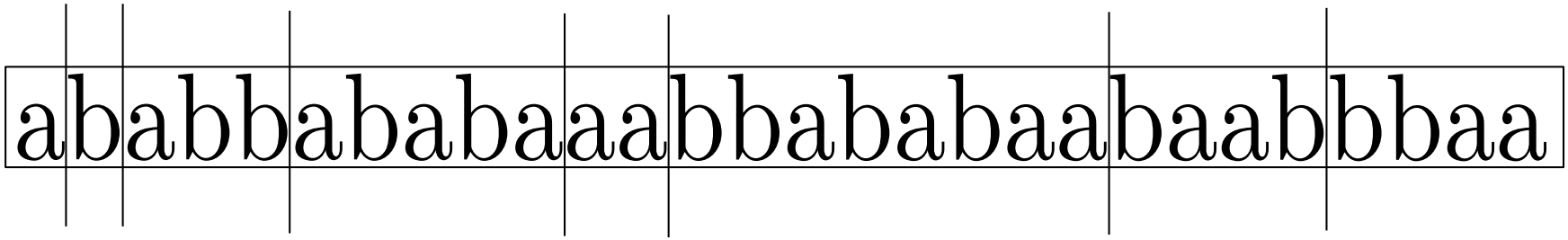
Example of LZ compression:

ababbababaaaabbbabababaabaabbbbaa

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.

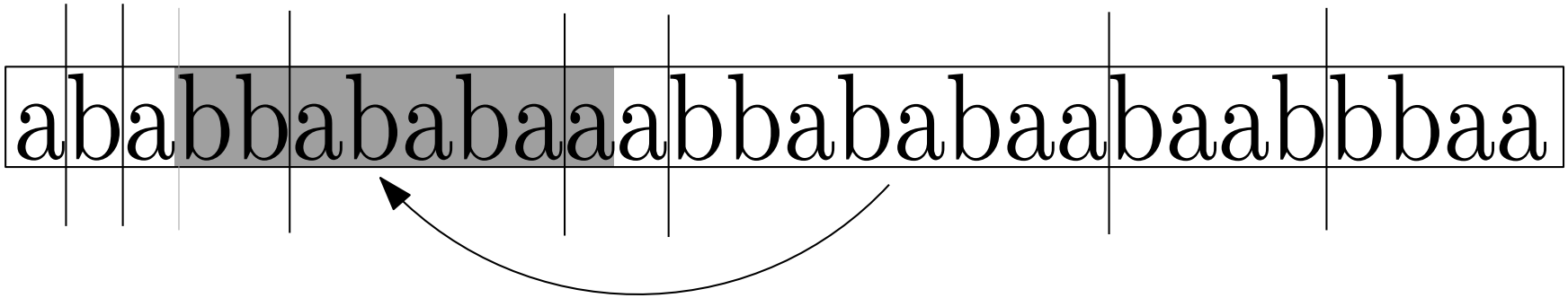
Example of LZ compression:



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.

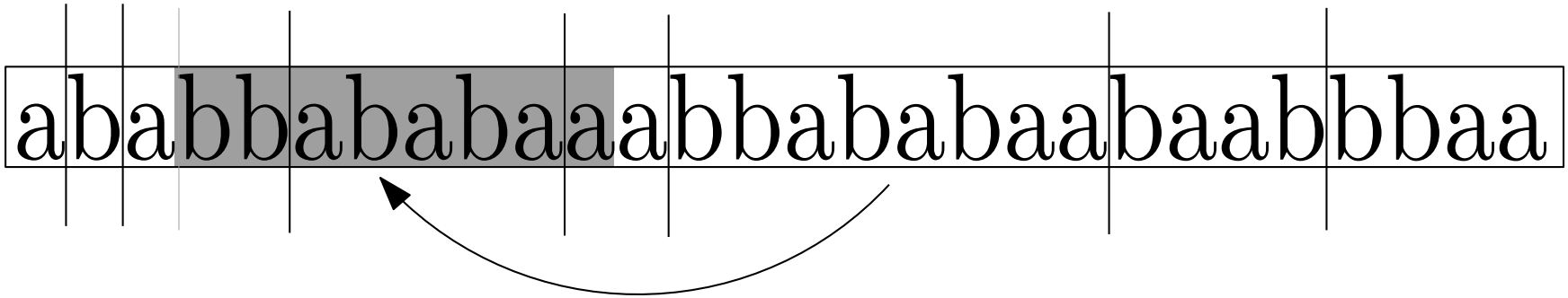
Example of LZ compression:



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.

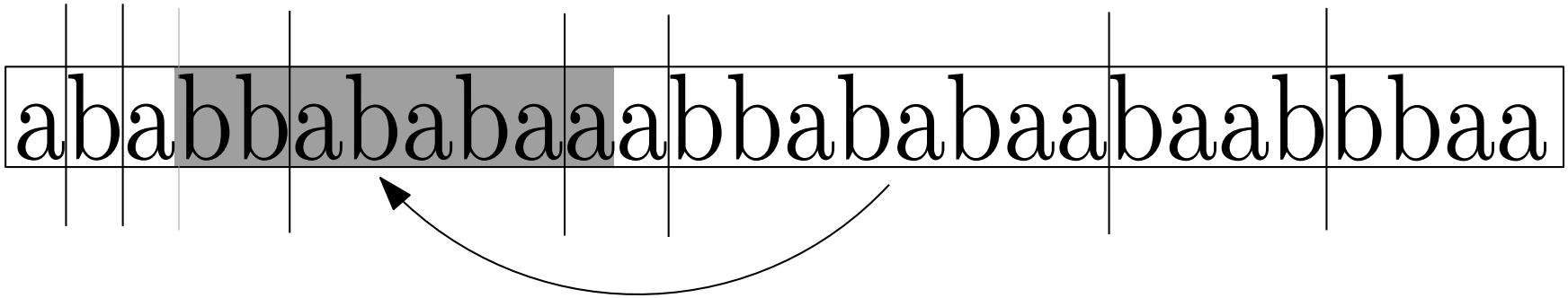
Example of LZ compression:



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 LPF$[1..n]$. LPF$[i]$ is the largest k such that $w[i..i+k-1]$ occurs starting somewhere on the left of i. More formally, $w[i..i+k-1] = w[i'..i'+k-1]$ with $i' < i$. Note that the previous occurrence is allowed to overlap with $w[i..i+k-1]$!

Having the LPF[1..n] table, we can easily output the self-referential LZ parse.

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

Message to take back home

Sometimes it pays off to reduce a more complicated problem.

Now we focus on computing all 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 that in constant time, remember?), 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 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 LPF$[n-1]$. Then remove $w[n-1..n]$ from the suffix array, and look at the predecessor/successor of $n-2$ to compute 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 current 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 neighbours 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 LCP($w[x..n], w[y..n]$) and LCP($w[y..n], w[z..n]$), Then, 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 LPF[i] can be computed in constant time per entry.

Surprisingly, the LZ factorization can be used to speed-up some string algorithms. A canonical example is detecting regularities, or simply squares.

Detecting squares

Given a word w, locate its substring $w[i..j]$ which is a square. A square is a word of the form xx.

Squares in everyday life

`tamtam`, `cancan`

Intuition

Lz factorization can help us to detect repetitions in our word: if some fragments repeats, the corresponding part of the parse should be short.

To formalize the intuition, look at the LZ parse $w[1..n] = b_1 b_2 \dots b_z$. Look at the leftmost square $w[i..j] = xx$ (leftmost means that j is smallest possible). Say that it ends inside b_k. Where can it begin?

Lemma

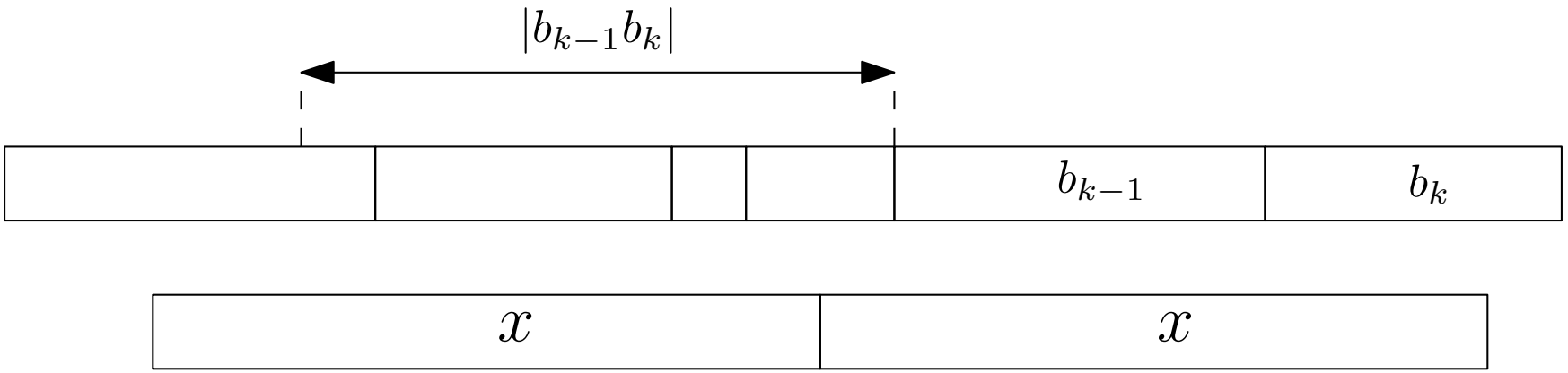
$$|w[i..j]| \leq 2|b_{k-1} b_k|$$

So, the leftmost square cannot begin too far to the left.

Lemma

$$|w[i..j]| \leq 2|b_{k-1}b_k|$$

Assume $|w[i..j]| > 2|b_{k-1}b_k|$ and draw a picture.



Whole b_{k-1} is inside the second x, so also inside the first x, so it occurs before. But b_{k-1} was chosen to be as long as possible, and it turns out that could be actually one character longer.

Using iterate over all possible k and try to use the lemma to detect the leftmost square ending in b_k. So, we need the following procedure.

Procedure

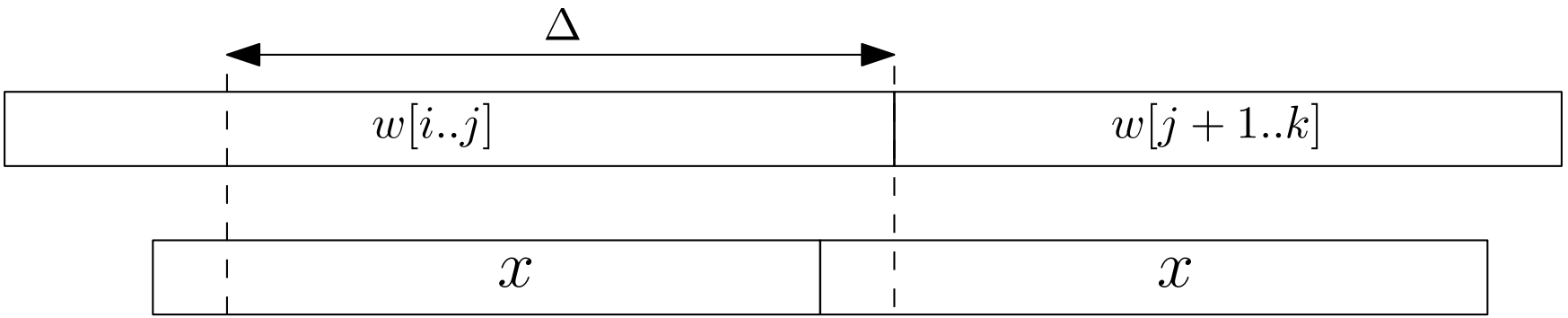
Given two consecutive fragments of w, say $w[i..j]w[j+1..k]$, detect a square ending in $w[j+1..k]$ and centered somewhere in $w[i..j]$ in $\mathcal{O}(|w[i..k]|)$ time.

Assuming that we know how to achieve such complexity, our total running time will be $\sum_k \mathcal{O}(2|b_{k-1}b_k|) = \mathcal{O}(\sum_k |b_k|) = \mathcal{O}(n)$.

Now let's try to implement the procedure.

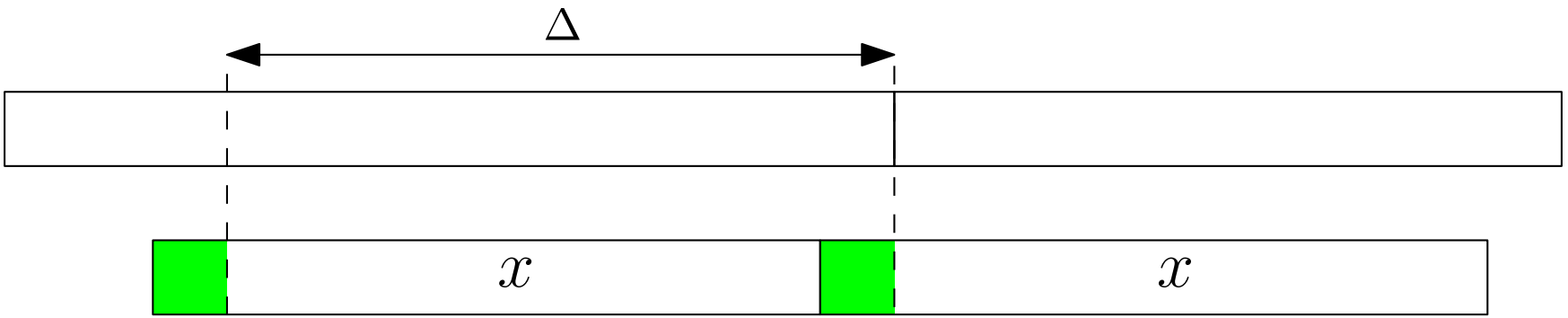
We guess $\Delta = |x|$. Then, both green/blue fragments are the same. With longest common prefix/suffix queries we can compute in constant time the longest such green/blue fragment. We check if $\ell + r \leq \Delta$. It's clear that such condition is necessary for a square to exists, and it turns out to be sufficient.

Now let's try to implement the procedure.



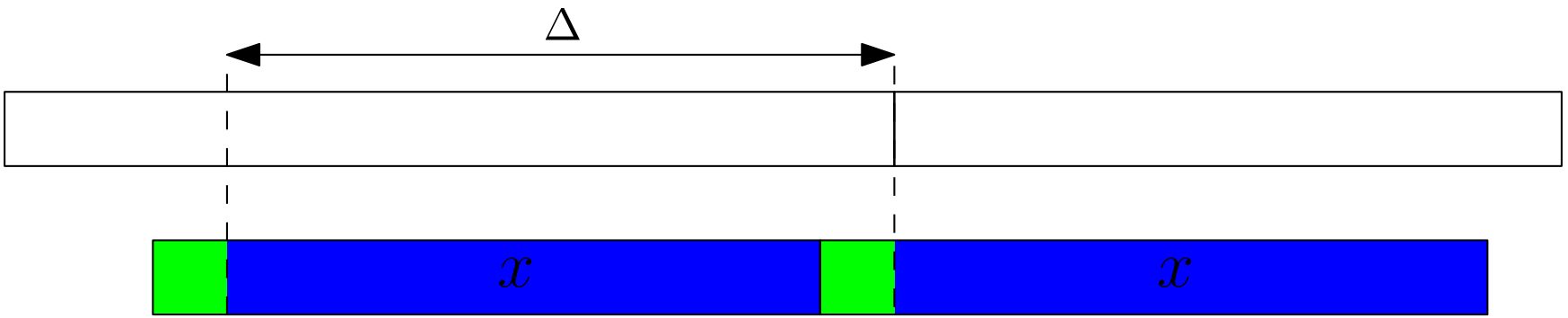
We guess $\Delta = |x|$. Then, both green/blue fragments are the same. With longest common prefix/suffix queries we can compute in constant time the longest such green/blue fragment. We check if $\ell + r \leq \Delta$. It's clear that such condition is necessary for a square to exists, and it turns out to be sufficient.

Now let's try to implement the procedure.



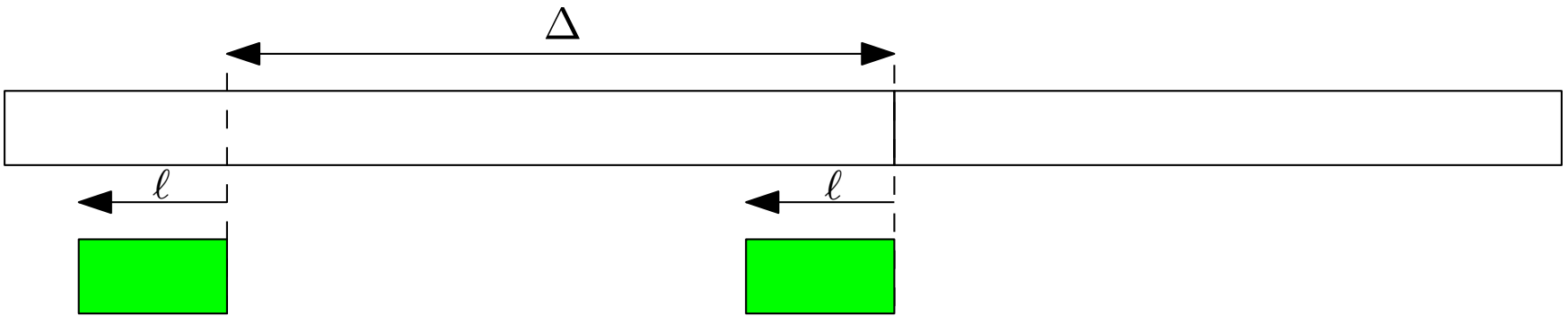
We guess $\Delta = |x|$. Then, both green/blue fragments are the same. With longest common prefix/suffix queries we can compute in constant time the longest such green/blue fragment. We check if $\ell + r \leq \Delta$. It's clear that such condition is necessary for a square to exists, and it turns out to be sufficient.

Now let's try to implement the procedure.



We guess $\Delta = |x|$. Then, both green/blue fragments are the same. With longest common prefix/suffix queries we can compute in constant time the longest such green/blue fragment. We check if $\ell + r \leq \Delta$. It's clear that such condition is necessary for a square to exists, and it turns out to be sufficient.

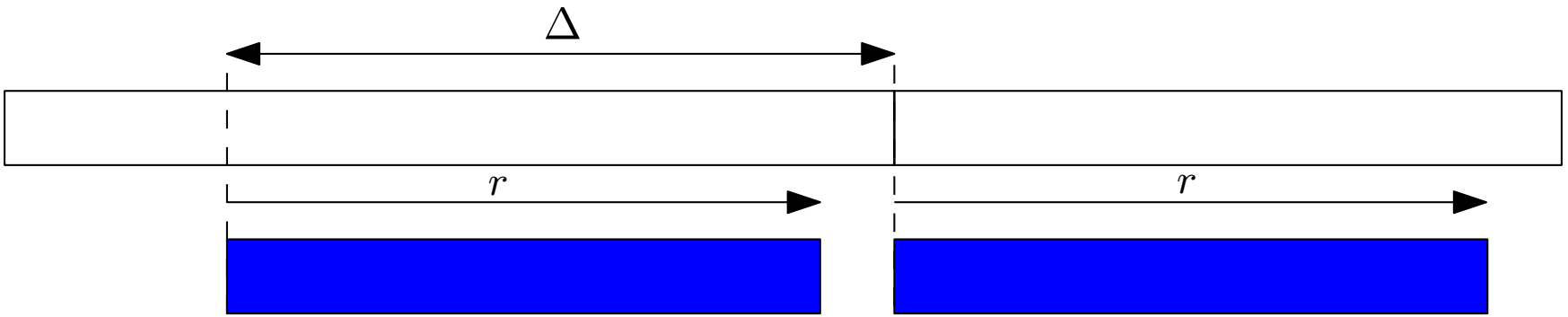
Now let's try to implement the procedure.



We guess $\Delta = |x|$. Then, both green/blue fragments are the same. With longest common prefix/suffix queries we can compute in constant time the longest such green/blue fragment.

We check if $\ell + r \leq \Delta$. It's clear that such condition is necessary for a square to exists, and it turns out to be sufficient.

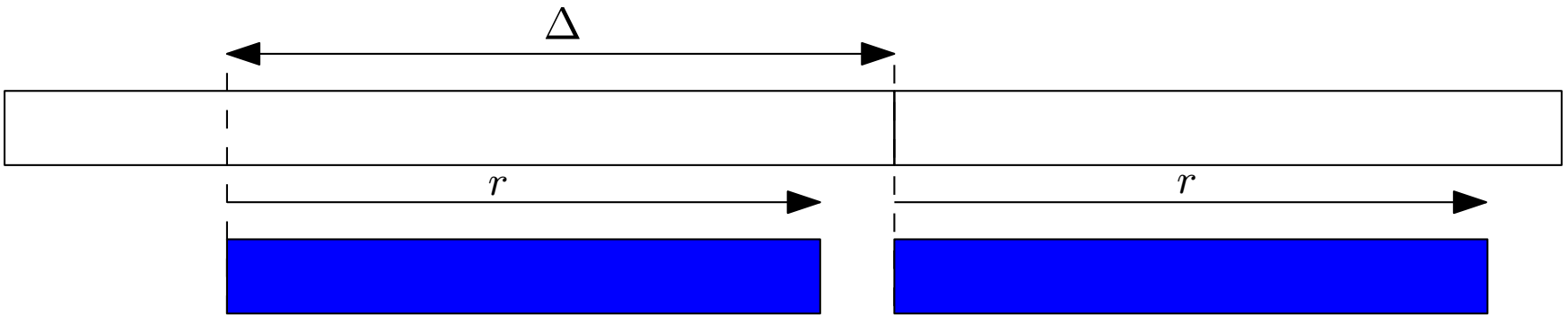
Now let's try to implement the procedure.



We guess $\Delta = |x|$. Then, both green/blue fragments are the same. With longest common prefix/suffix queries we can compute in constant time the longest such green/blue fragment.

We check if $\ell + r \leq \Delta$. It's clear that such condition is necessary for a square to exists, and it turns out to be sufficient.

Now let's try to implement the procedure.



We guess $\Delta = |x|$. Then, both green/blue fragments are the same. With longest common prefix/suffix queries we can compute in constant time the longest such green/blue fragment. We check if $\ell + r \leq \Delta$. It's clear that such condition is necessary for a square to exists, and it turns out to be sufficient.

So, we first find the LZ factorization (in linear time), then use the procedure z times. The total running time is linear, assuming that we can compute the longest common prefix/suffix between any two fragments in constant time (which we know how to do).

Theorem

The leftmost square can be found in linear time.