max planck institut
informatik

Faster searching and RMQ in constant time

Algorithms on Strings

Paweł Gawrychowski

June 12, 2013

Outline

Recall that 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

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

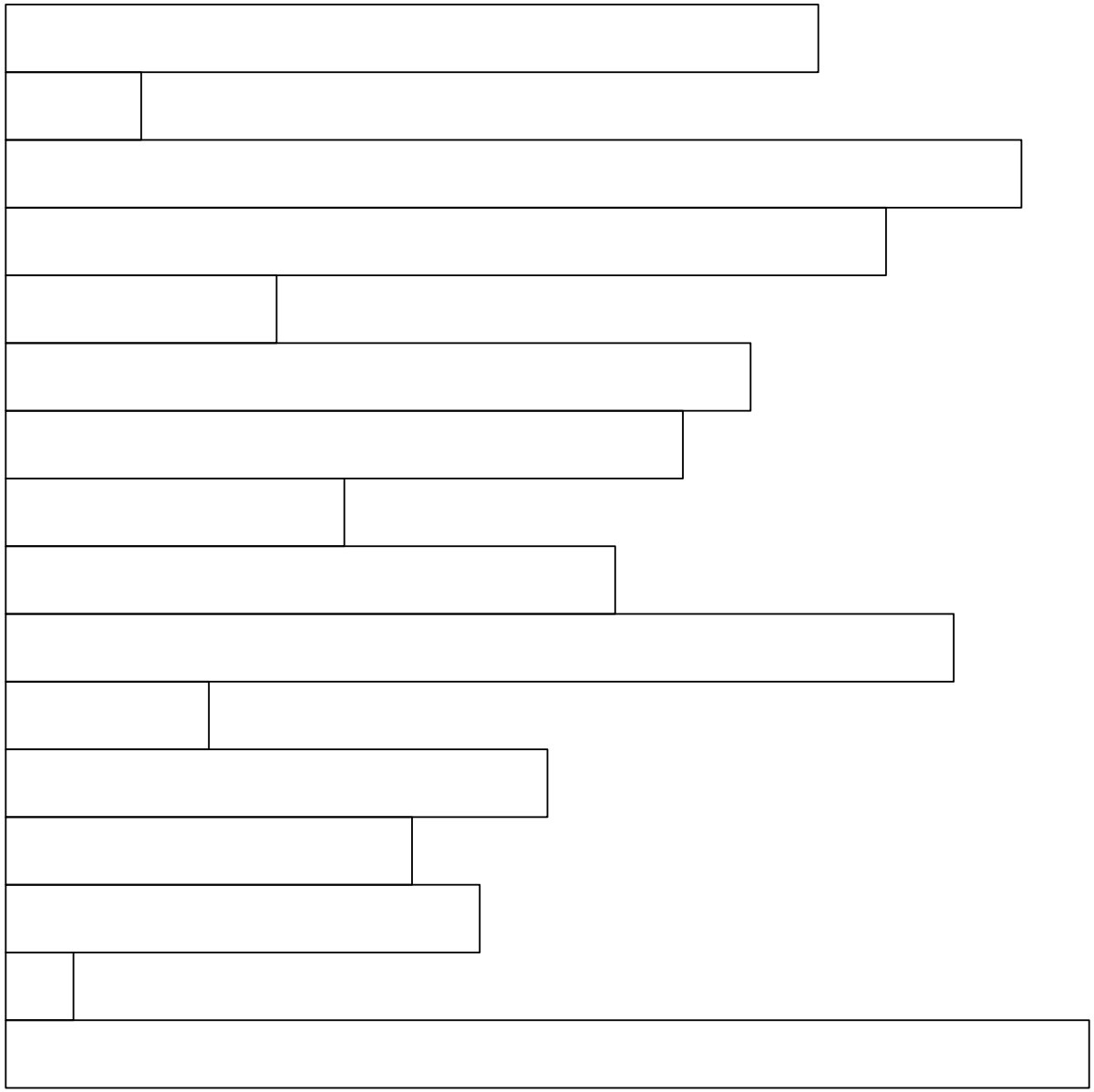
Binary search uses $\log n$ iterations, but each of them might cost even $\Omega(m)$ operations! Hence the whole procedure is $\mathcal{O}(m \log n)$.

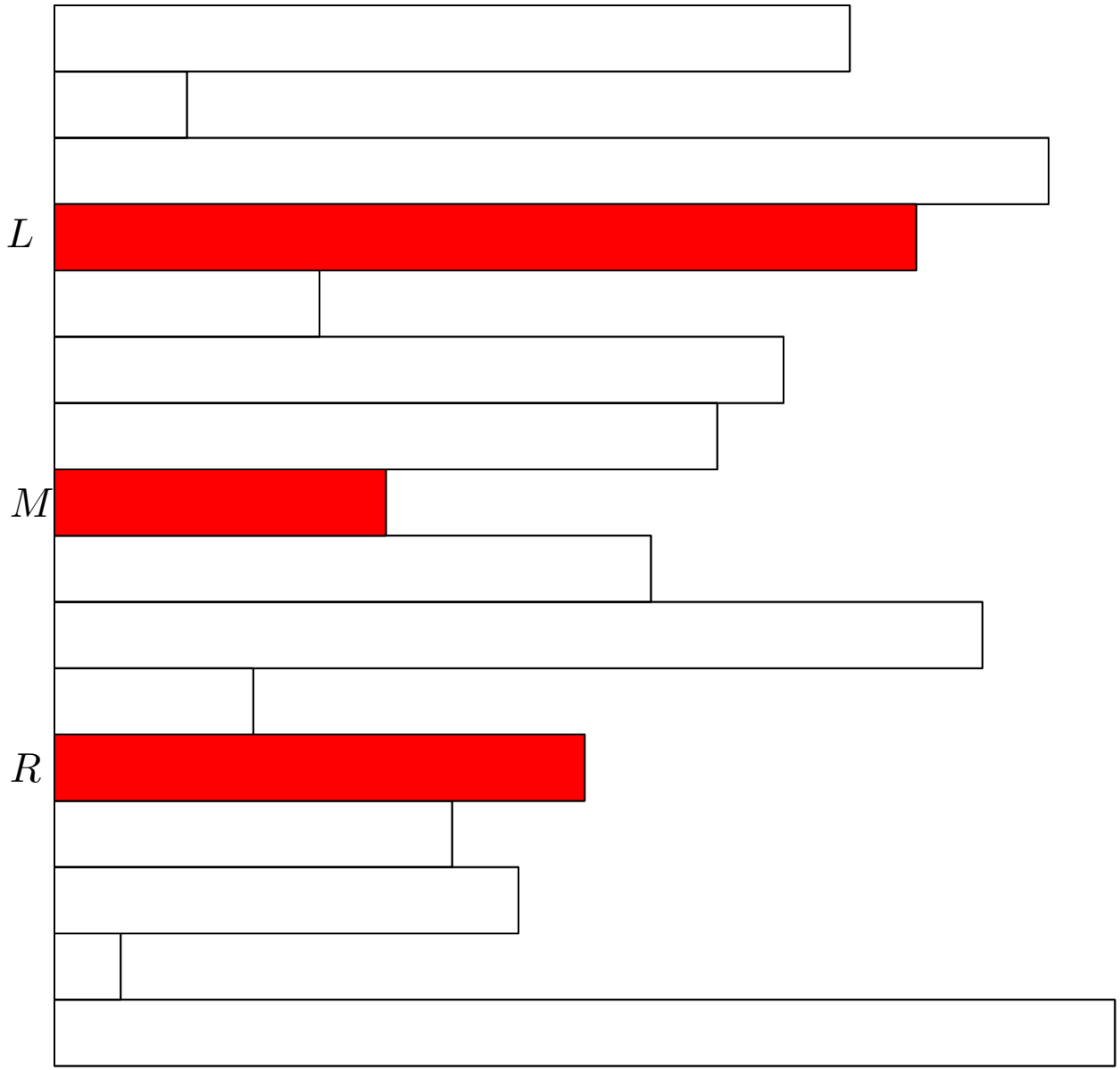
Now the question is whether we can do better. It seems that we are wasting lots of time comparing very similar blocks of texts again and again. That is not cool!

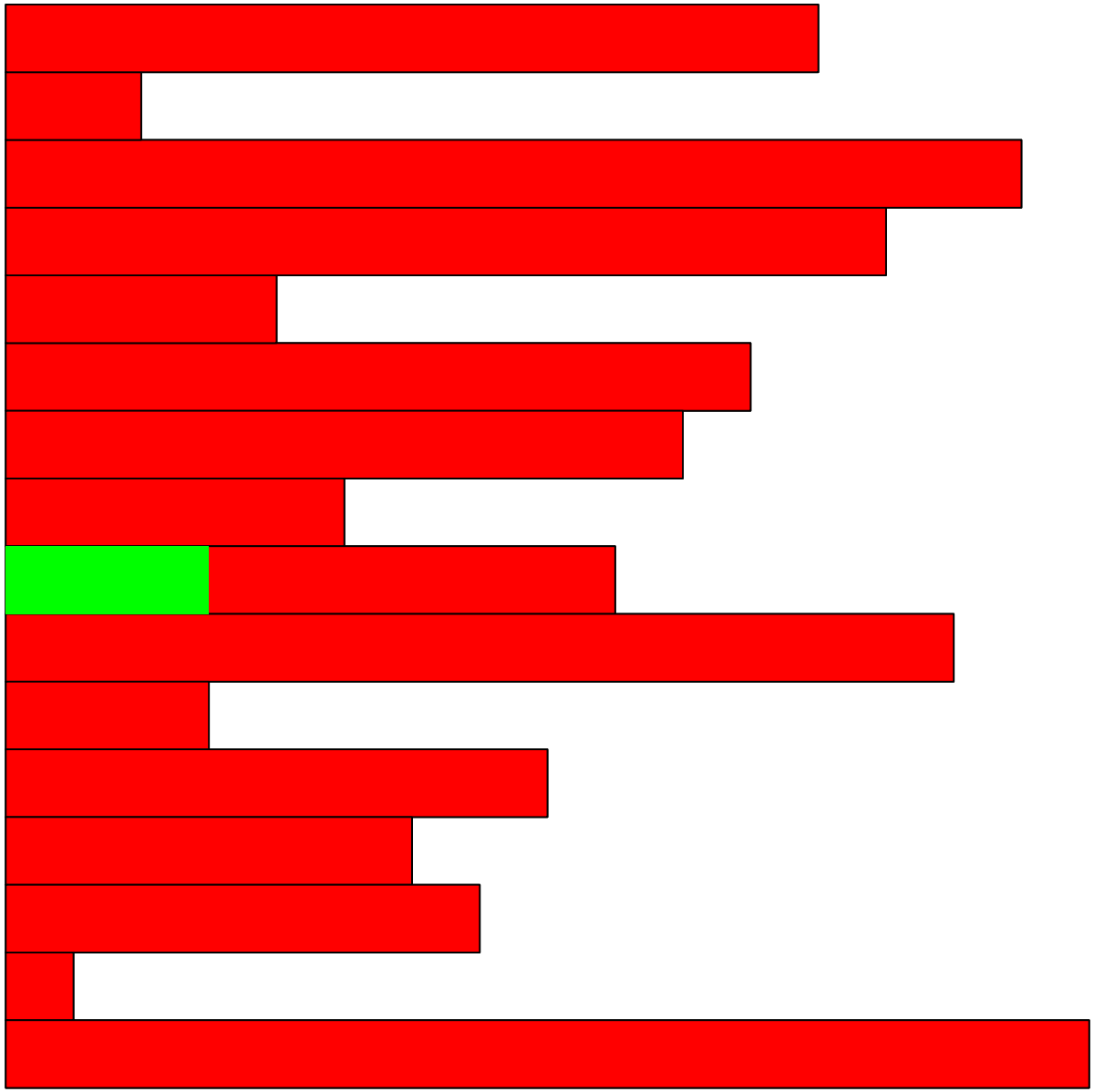
lcp for the rescue

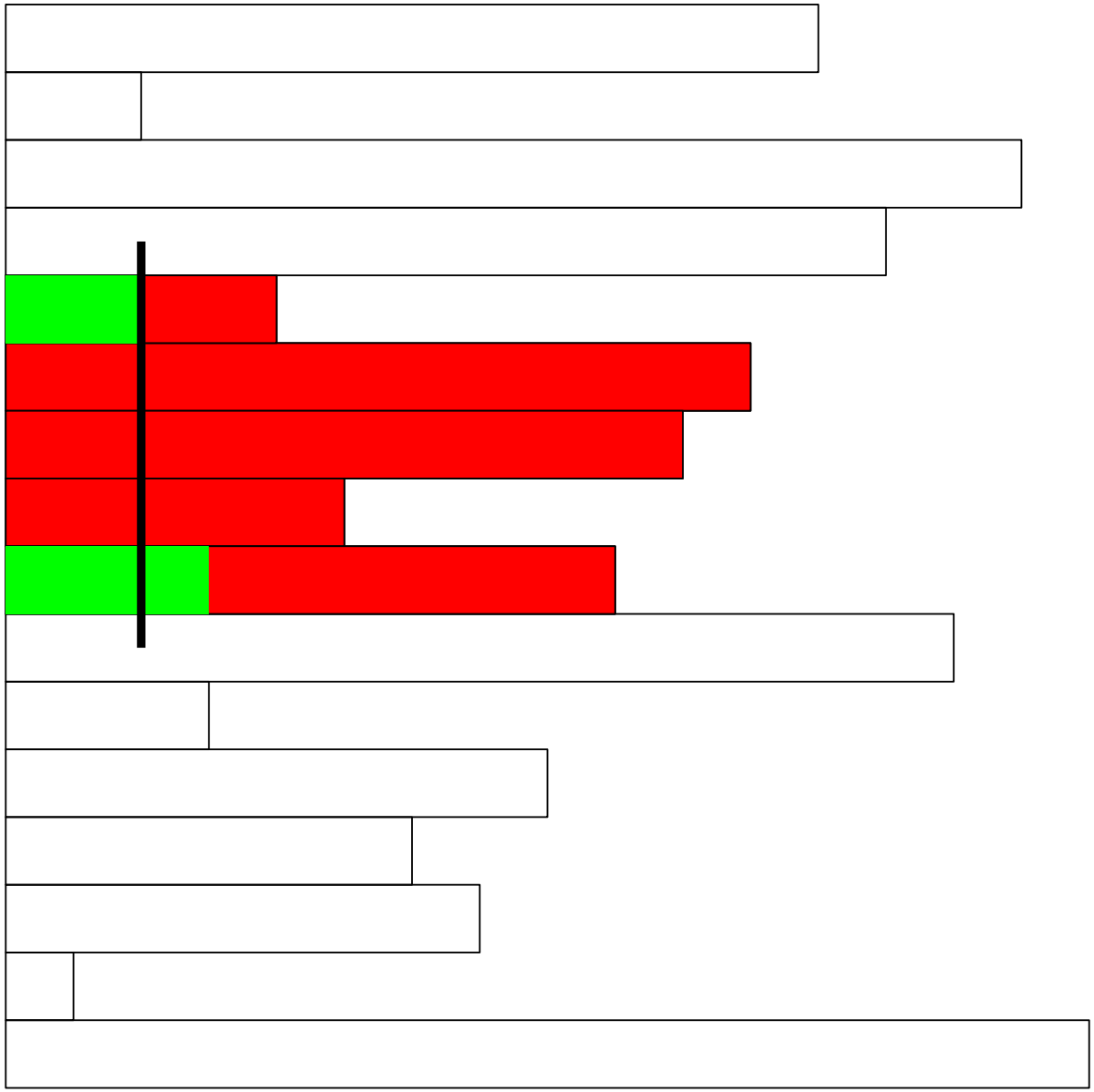
Recall that $\mathrm{lcp}(i,j)$ is the longest common prefix of the suffixes $w[i..n]$ and $w[j..n]$. Last week you have seen how to compute $\mathrm{lcp}[i]$, and two weeks ago we observed that computing $\mathrm{lcp}(i,j)$ reduces to the so-called Range Minimum Query on the $\mathrm{lcp}[i]$ array.

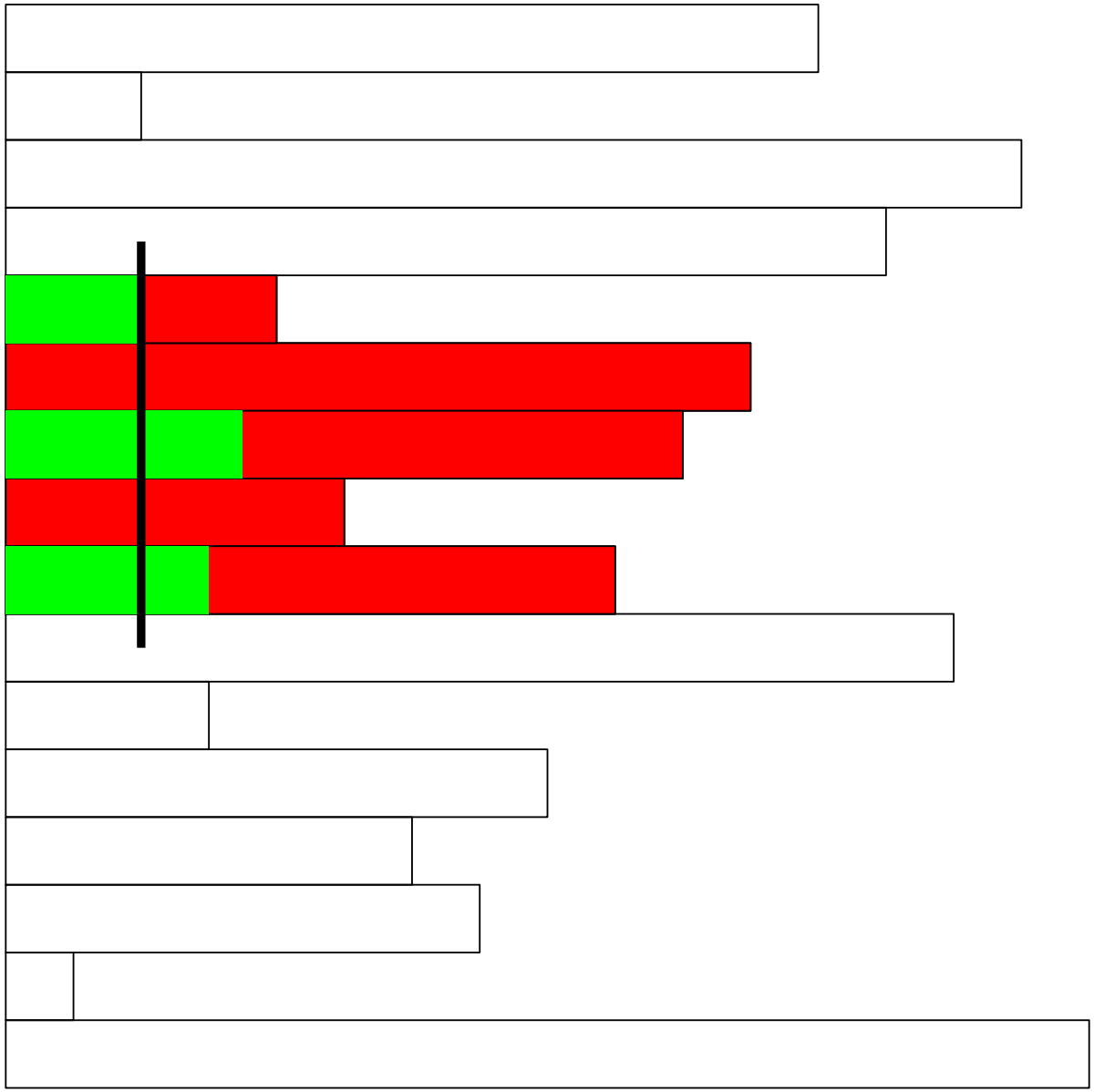
For the time being assume that we know how to answer the RMQ queries on any array in constant time. Then we can compute any $\mathrm{lcp}(i,j)$ in constant time. Can this help us to speed up the binary searching?




L
M
R



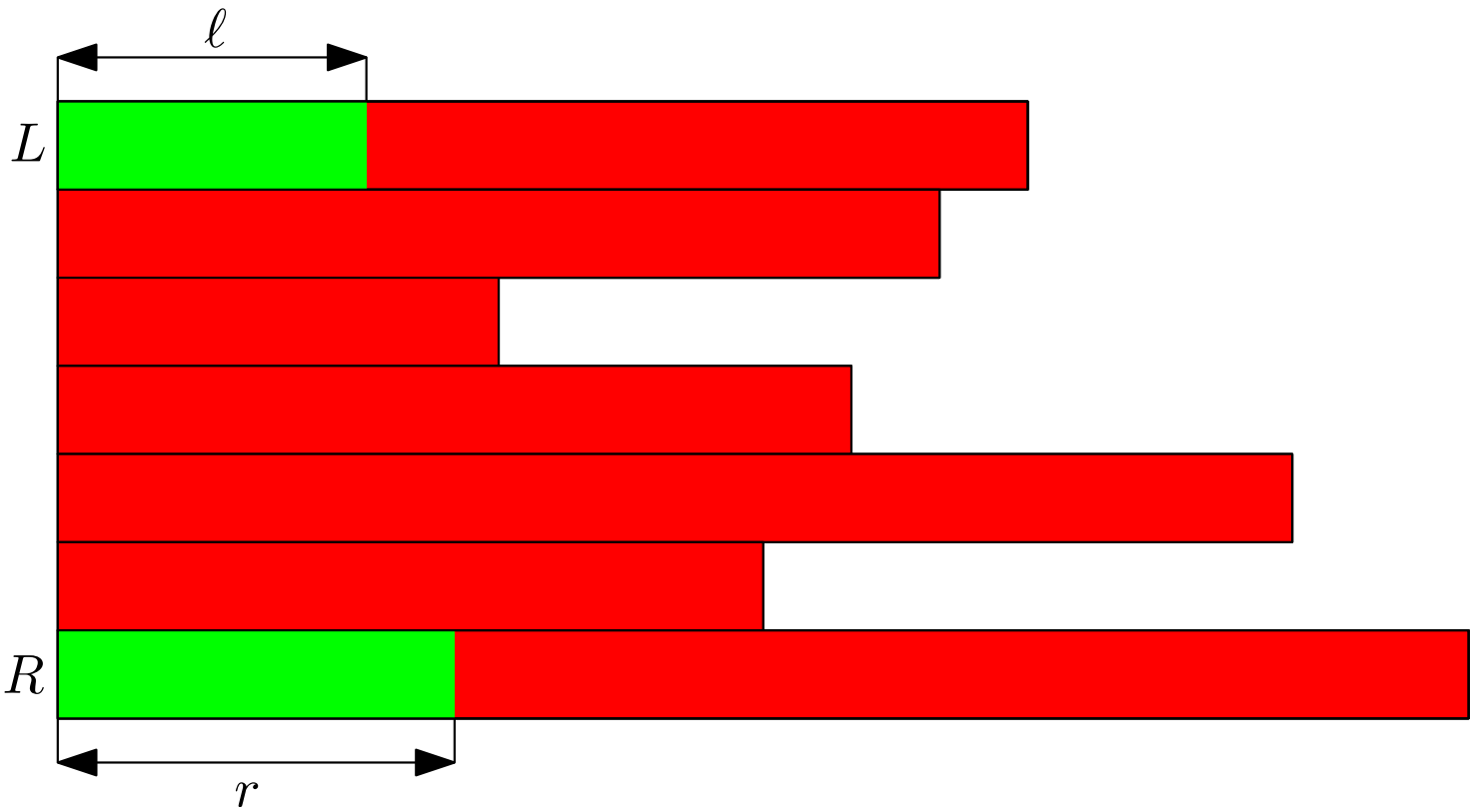


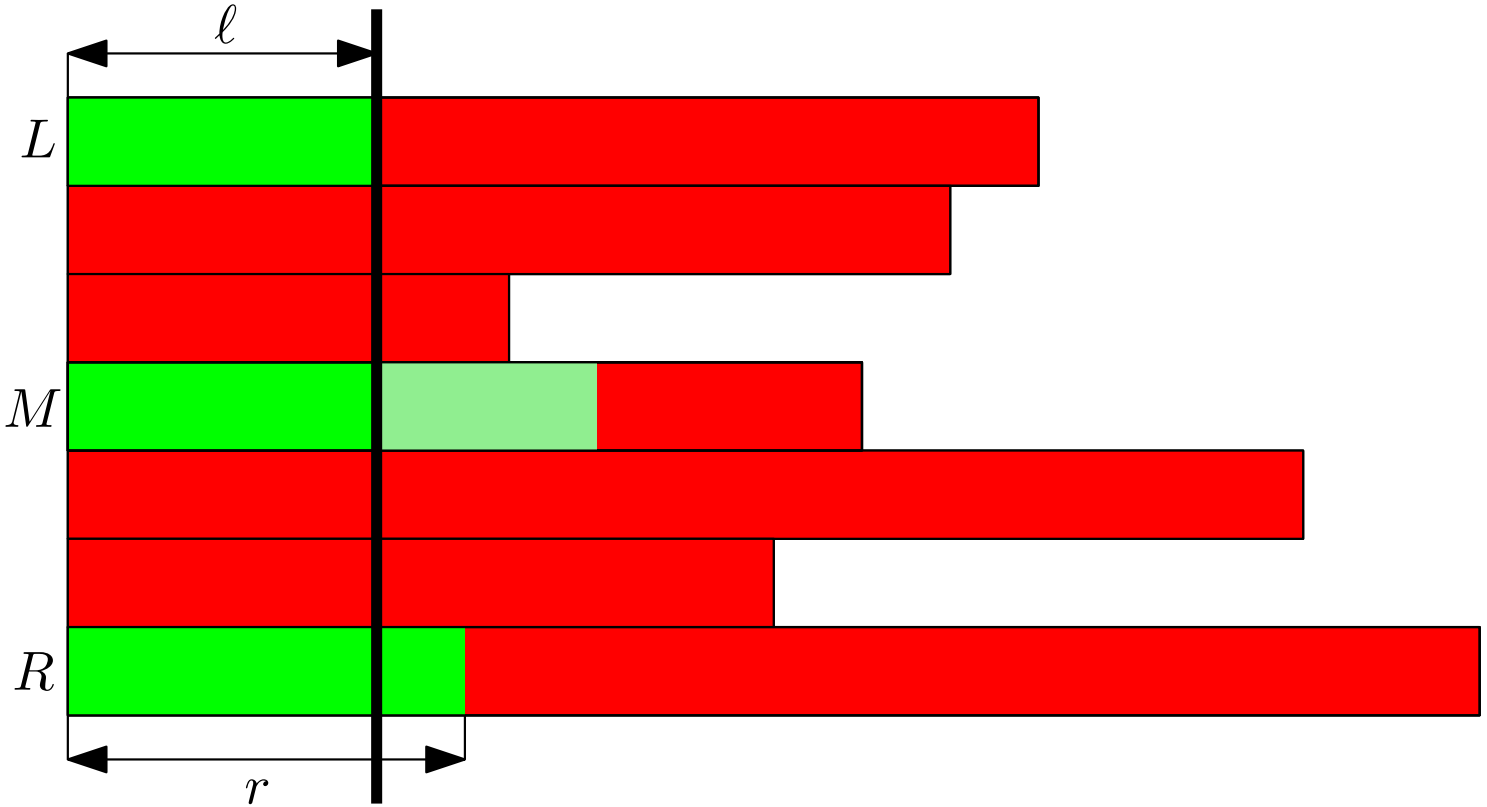


Invariant

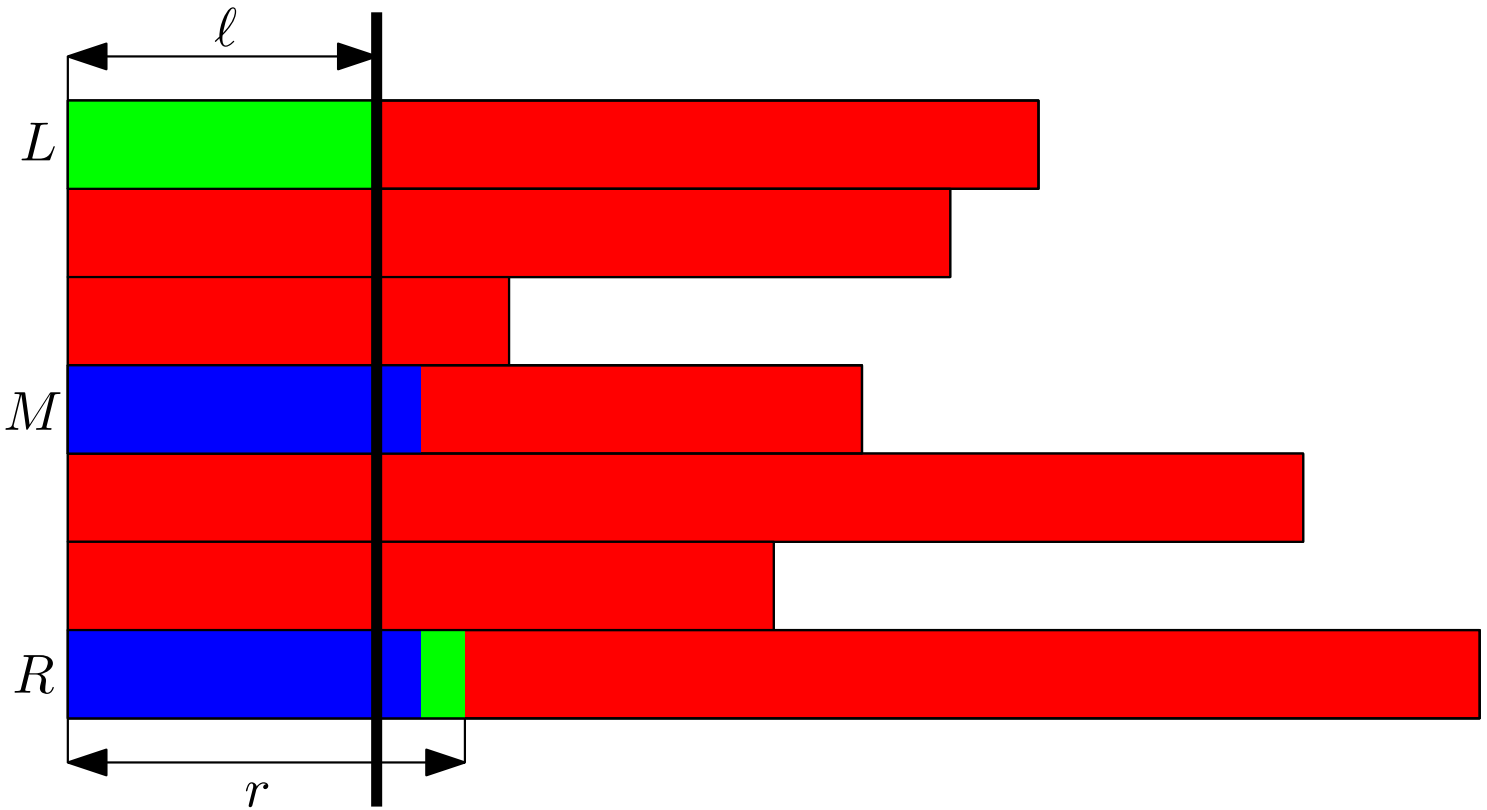
We maintain a range $[L, R]$ such that the answer is somewhere inside, and we know the longest common prefix of $SA[L]$ and p, and $SA[R]$ and p.

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.

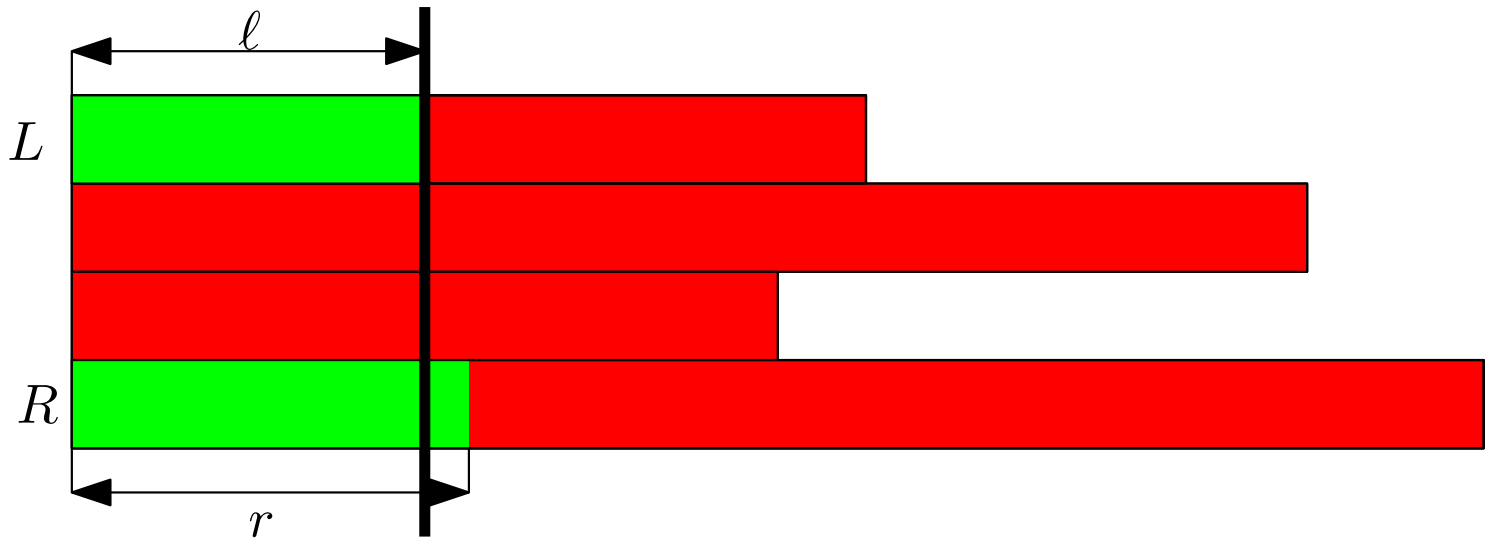

ℓ
L
R
r



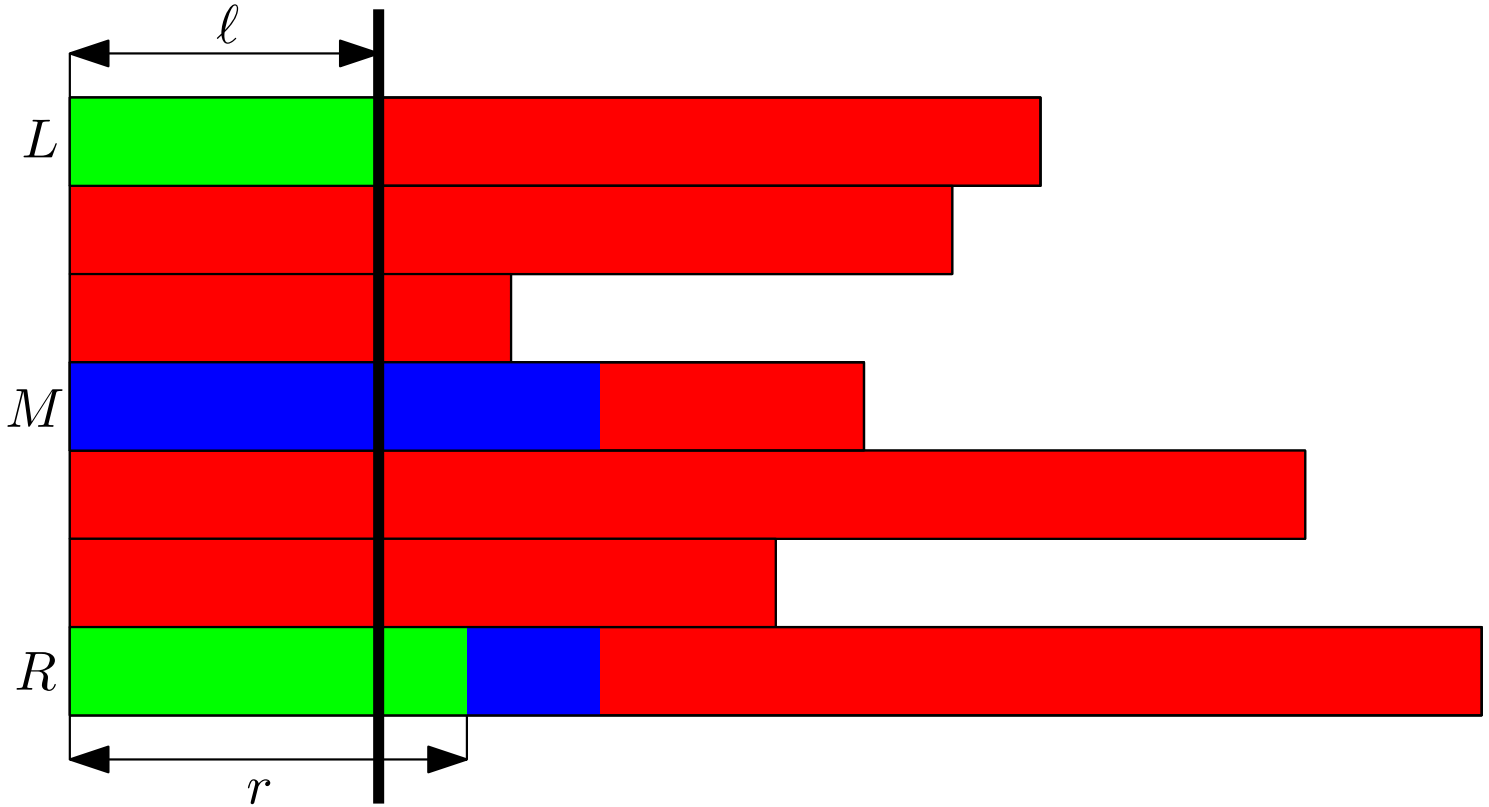
Look at lcp($SA[M], SA[R]$).



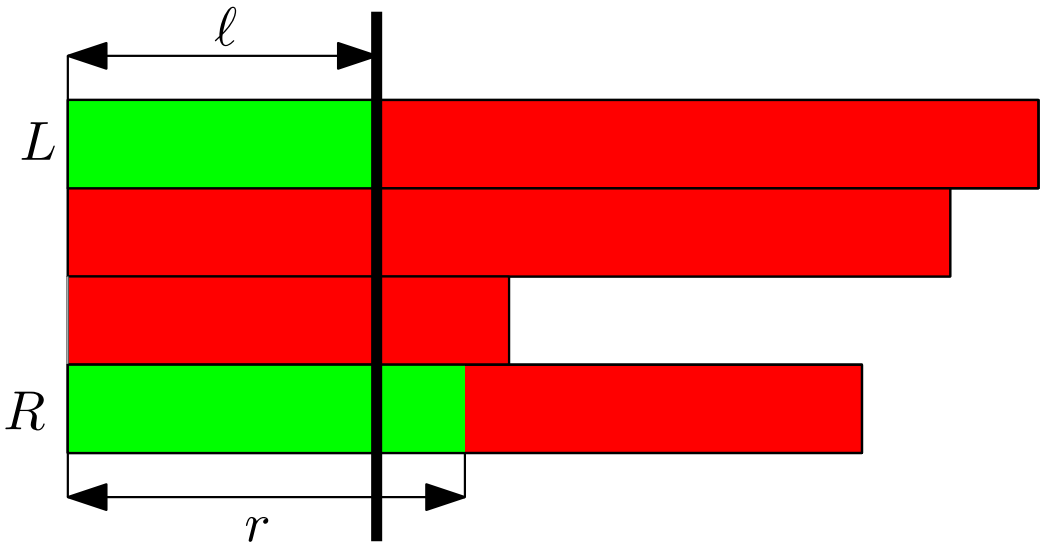
If $\mathrm{lcp}(SA[M], SA[R]) < r$, set $L = M$ and $\ell = \mathrm{lcp}(SA[M], SA[R])$.



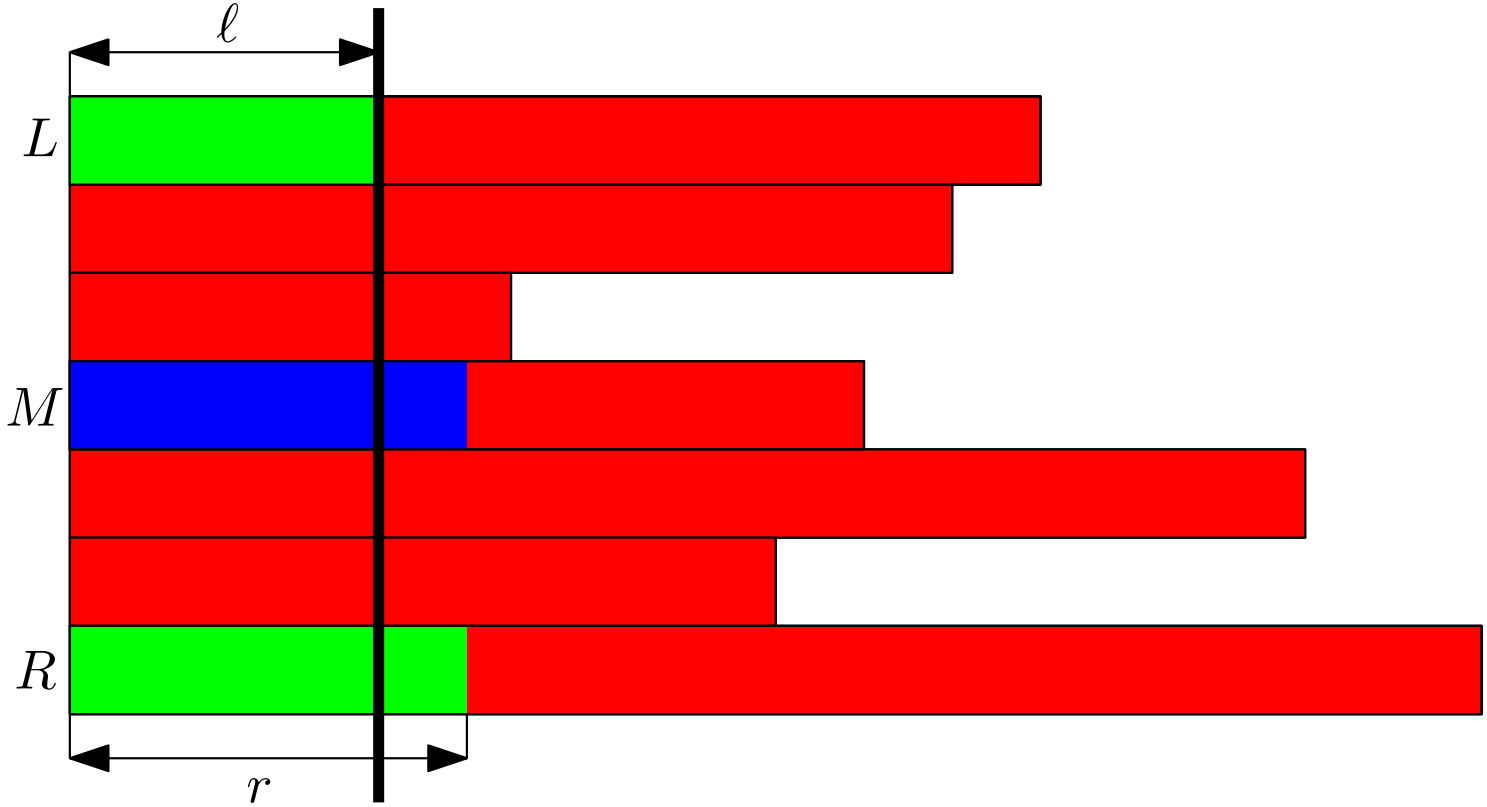
If $\mathrm{lcp}(SA[M], SA[R]) < r$, set $L = M$ and $\ell = \mathrm{lcp}(SA[M], SA[R])$.



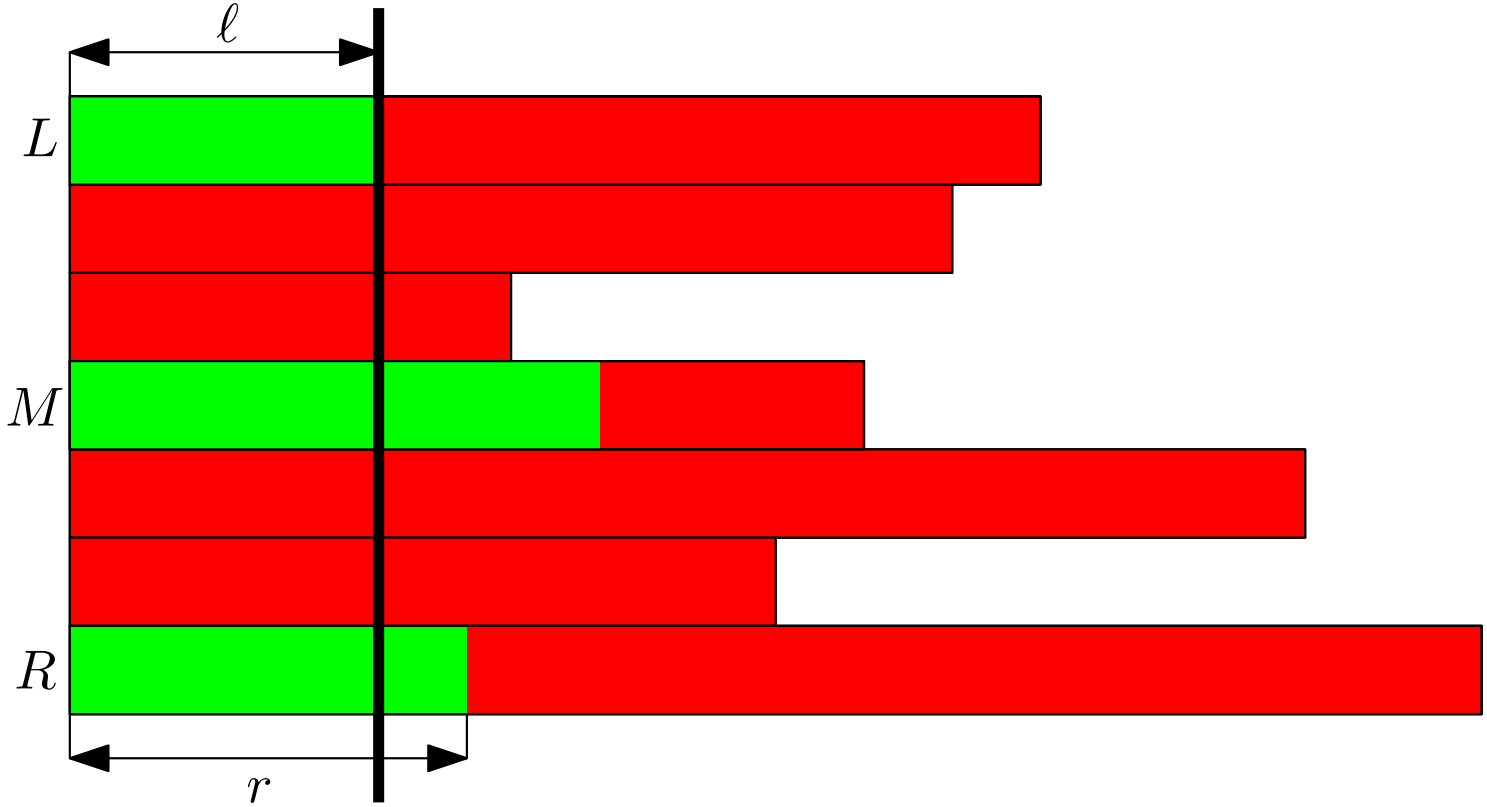
If $\mathrm{lcp}(SA[M], SA[R]) > r$, set $R = M$ and keep old ℓ and r.



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 m. We have two cases:

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

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

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

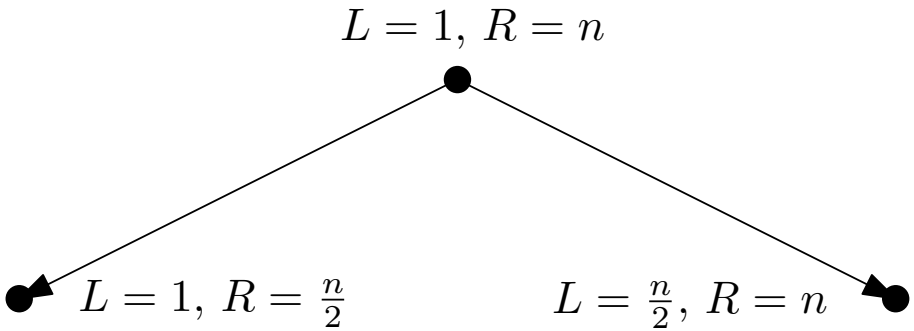
It follows that the sum of $\mathcal{O}(m - r)$ over all steps of the procedure is just $\mathcal{O}(m)$ in the worst possible case. We additionally spent $\mathcal{O}(1)$ time per step to look at $SA[M]$, hence the total complexity is $\mathcal{O}(m + \log n)$.

Recall that we assumed that computing any lcp(i, j) takes constant time. While it can be done (as we will soon see), that's an overkill. Do we really need to compute any such value?

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

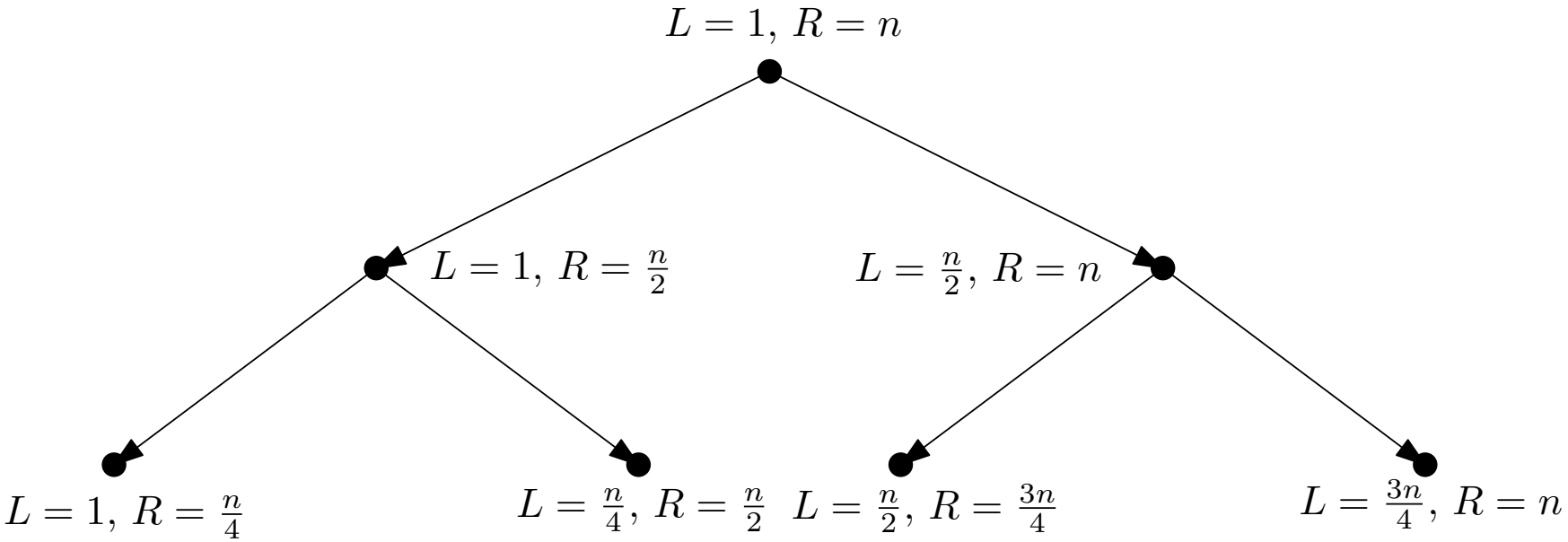
Each node of the recursion tree generates just two values lcp$(SA[L], SA[M])$ and lcp$(SA[M], SA[R])$ to be computed. Hence we have just $\mathcal{O}(n)$ values in total!

Recall that we assumed that computing any lcp(i, j) takes constant time. While it can be done (as we will soon see), that's an overkill. Do we really need to compute any such value?



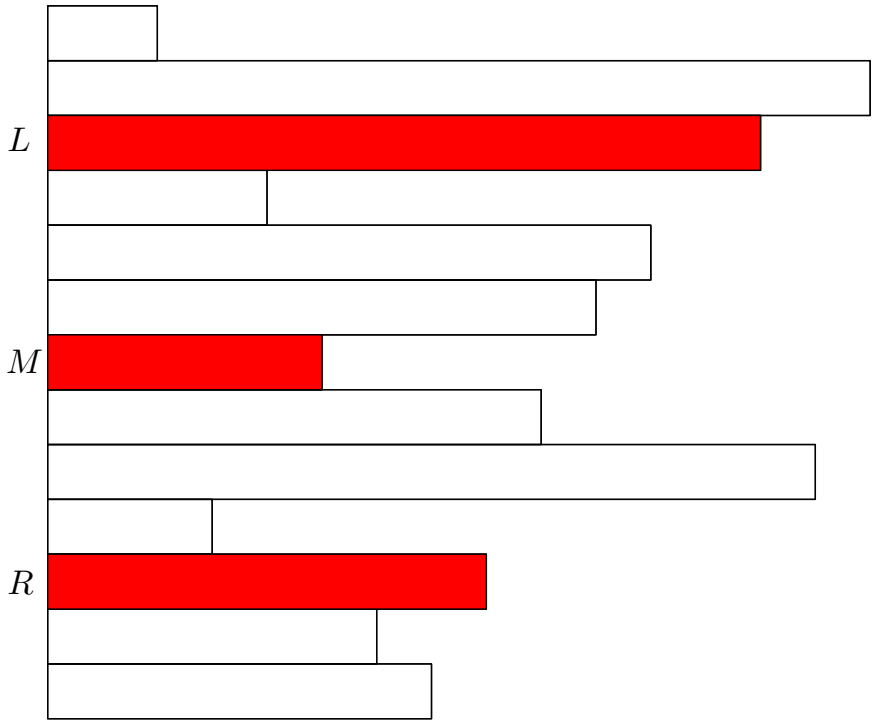
Each node of the recursion tree generates just two values lcp$(SA[L], SA[M])$ and lcp$(SA[M], SA[R])$ to be computed. Hence we have just $\mathcal{O}(n)$ values in total!

Recall that we assumed that computing any lcp(i, j) takes constant 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 lcp$(SA[L], SA[M])$ and 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

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

Proof: see the whiteboard.

Even though we have shown that having just the lcp[i] array allows us to execute the binary search efficiently, being able to answer any lcp(i, j) would be great. Recall that we were able to reduce the question to the s-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.

For starters, 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 constant time after $\mathcal{O}(n \log n)$ time and space preprocessing.

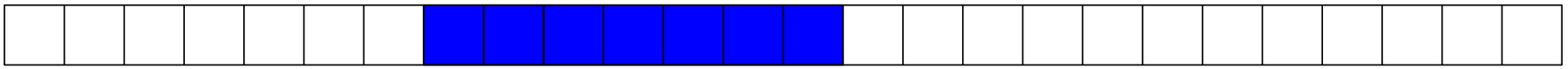
To prove the lemma, we will (again) apply the simple-yet-powerful doubling paradigm. For each $k = 0, 1, \ldots, \log n$ construct a table B_k.

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

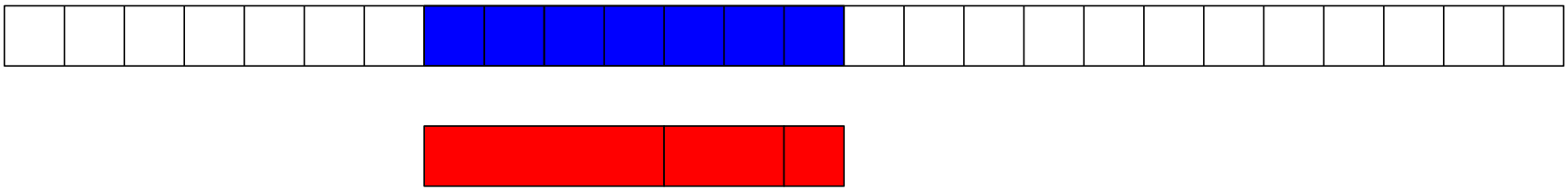
How? Well, $B_0[i] = A[i]$, and $B_{k+1}[i] = \min(B_k[i], B_k[i+2^k])$. Hence we can easily answer a query concerning a fragment of length that is a power of 2. But, unfortunately, not all numbers are powers of 2...

...or are they?

...or are they?

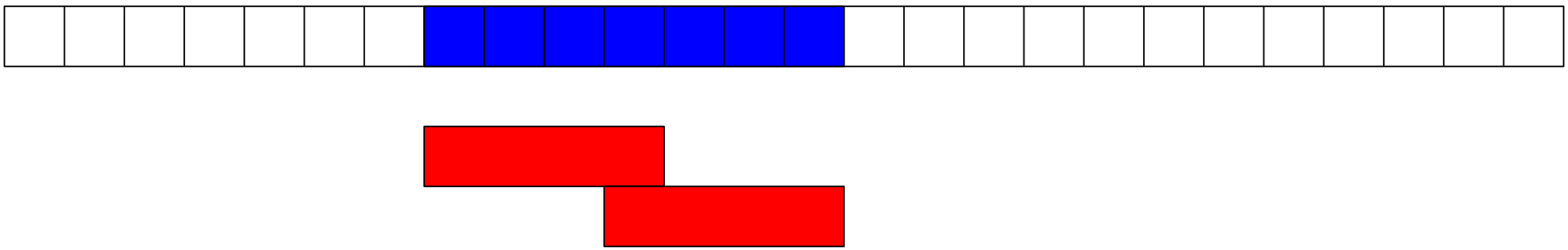


...or are they?



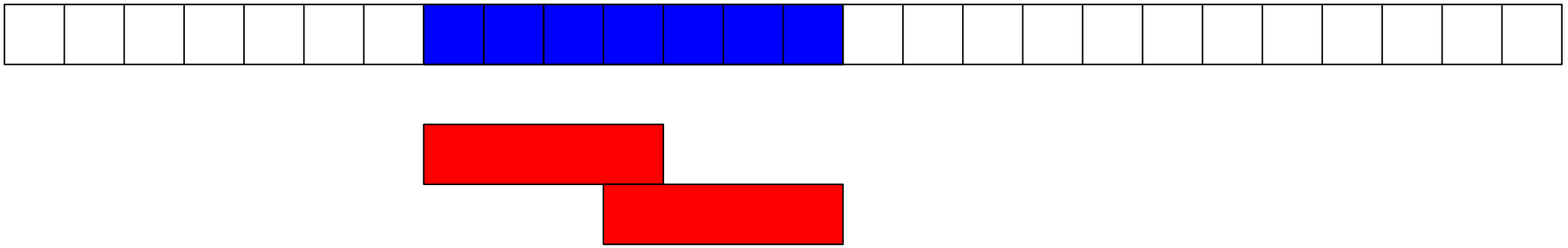
Any query can be split into at most log n power-of-two queries.

...or are they?



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

...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 the two power-of-two queries, 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.

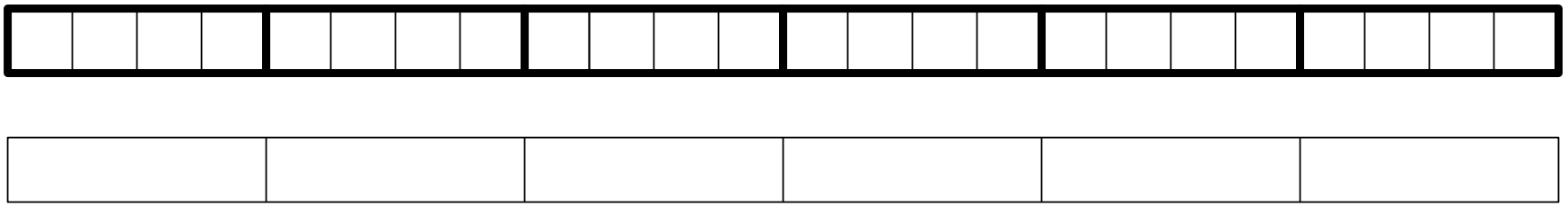
We apply another simple-yet-powerful paradigm: micro-macro decomposition. Chop the input array into blocks of length $b = \log n$.

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 paradigm: micro-macro decomposition. Chop the input array into blocks of length $b = \log n$.

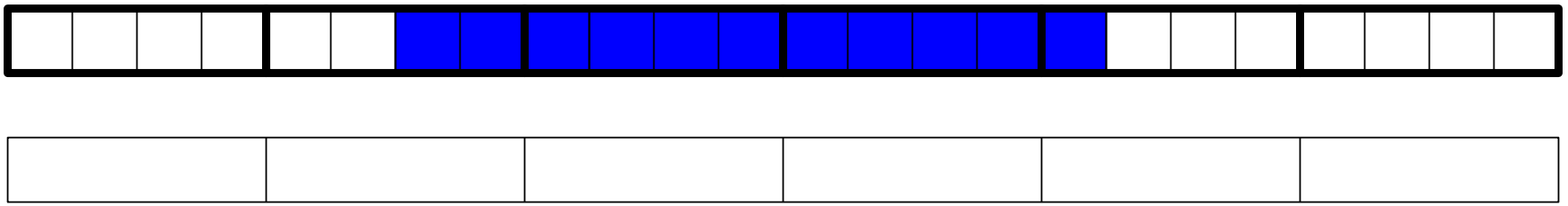


Construct a new array A', where $A'[i] = \min\{A[ib+1], A[ib+2], \ldots, A[(i+1)b]\}$. Build the previously described structure for A'.

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 paradigm: micro-macro decomposition. Chop the input array into blocks of length $b = \log n$.

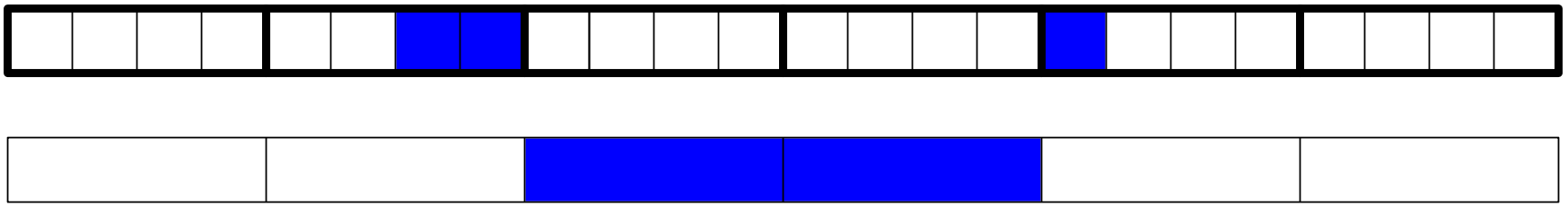


Construct a new array A', where $A'[i] = \min\{A[ib+1], A[ib+2], \ldots, A[(i+1)b]\}$. Build the previously described structure for A'.

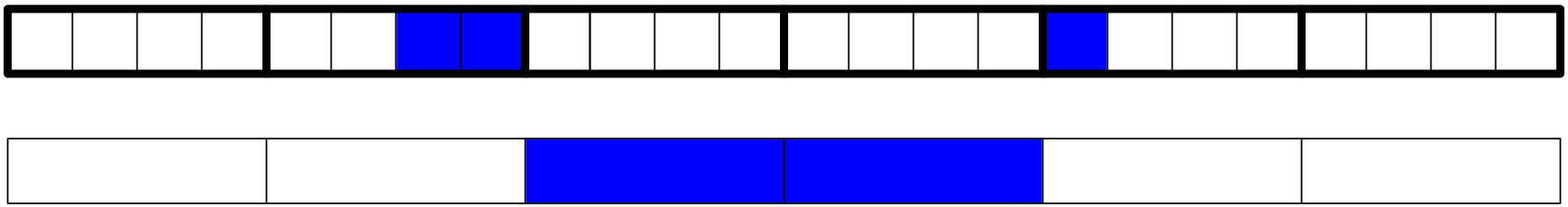
Lemma

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

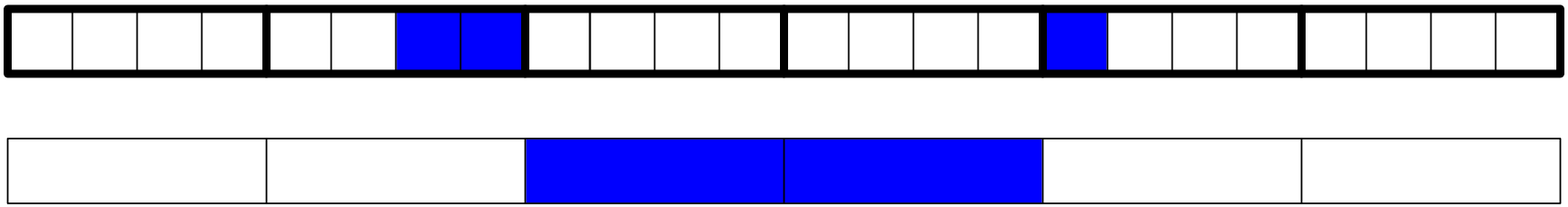
We apply another simple-yet-powerful paradigm: micro-macro decomposition. Chop the input array into blocks of length $b = \log n$.



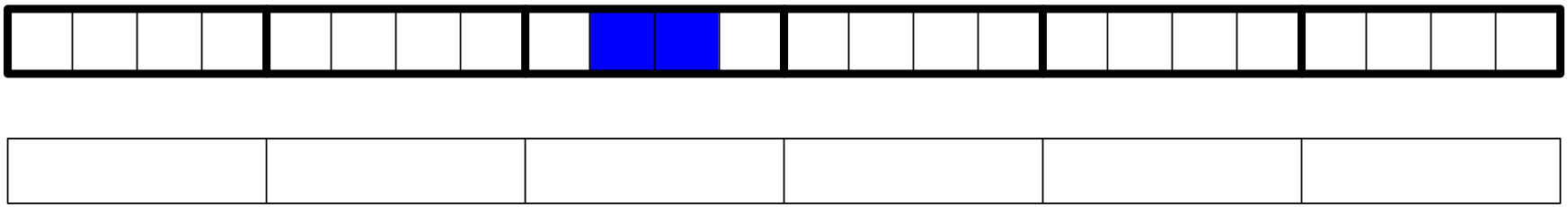
Construct a new array A', where $A'[i] = \min\{A[ib+1], A[ib+2], \ldots, A[(i+1)b]\}$. Build the previously described structure for A'.



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 constant time.
Unfortunately, life is not that simple.
But the only case when we cannot answer a query in constant 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 constant time.
Unfortunately, life is not that simple.
But the only case when we cannot answer a query in constant 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 constant time.
Unfortunately, life is not that simple.
But the only case when we cannot answer a query in constant 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: constant query and linear 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 that for a very restricted case, when $|A[i+1] - A[i]| \leq 1$.

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 constant time.