String indexing in the Word RAM model, part 1

Paweł Gawrychowski

University of Wrocław

Sometimes we use the Turing machine model, where we have a fixed number of tapes consisting of binary cells, and the only thing we can do is moving the heads. But this is not really how real computers are built, and we would like to work in a more realistic model.

Word RAM	Memory is divided into cells of size $w \geq \log n$ called **words**. There is a fixed set of $\mathcal{O}(1)$ time C-style operations, one of them being indirect addressing, so given a word containing x, we can access the cell x (not the case in the pointer machine model!). The input consists of numbers stored in single words.
AC^0 RAM	All operations must be implemented by constant-depth, unbounded fan-in, polynomial-size (in w) circuits. No multiplication.
Practical RAM	Just addition, shift, and bitwise boolean operations.
RAMBO RAM	Bits in different words may overlap.
Cell probe model	We only pay for accessing cells. The computation itself is free and the model is nonuniform. Good for lower bounds!

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 *bbabababababaab*.

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 $s[i] < t[i]$.

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 $s[i] < t[i]$.

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.

$w = \texttt{mississippi}$

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

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}.

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 $T(n) = \mathcal{O}(n)$.

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 $T(n) = \mathcal{O}(n)$.

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 sort S_0 by replacing a 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 sort S_0 by replacing a 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 sort S_0 by replacing a 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 sort S_0 by replacing a 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.

$W = \texttt{mississippi}$

$SA[1] =$	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 $\mathrm{lcp}(8, 2)$?

$W = \texttt{mississippi}$

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

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

Lemma

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

So what?

So, assuming that we know $\text{lcp}[i]$, computing any $\text{lcp}(i,j)$ 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

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

So what?

So, assuming that we know $\text{lcp}[i]$, computing any $\text{lcp}(i, j)$ 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 $k = SA^{-1}[i]+1$, $SA^{-1}[i]+2$, $\ldots$, $SA^{-1}[j]$, 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 $\mathrm{lcp}[2], \mathrm{lcp}[3], \ldots, \mathrm{lcp}[n]$?

Kasai et al. 2001

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

The procedure uses the following observation

$$\mathrm{lcp}[SA^{-1}[i]] - 1 \leq \mathrm{lcp}[SA^{-1}[i+1]]$$

See the problem set.

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

Kasai et al. 2001

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

The procedure uses the following observation

$$\mathrm{lcp}[SA^{-1}[i]] - 1 \leq \mathrm{lcp}[SA^{-1}[i+1]]$$

See the problem set.

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

Kasai et al. 2001

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

The procedure uses the following observation

$$\mathrm{lcp}[SA^{-1}[i]] - 1 \leq \mathrm{lcp}[SA^{-1}[i+1]]$$

See the problem set.

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.

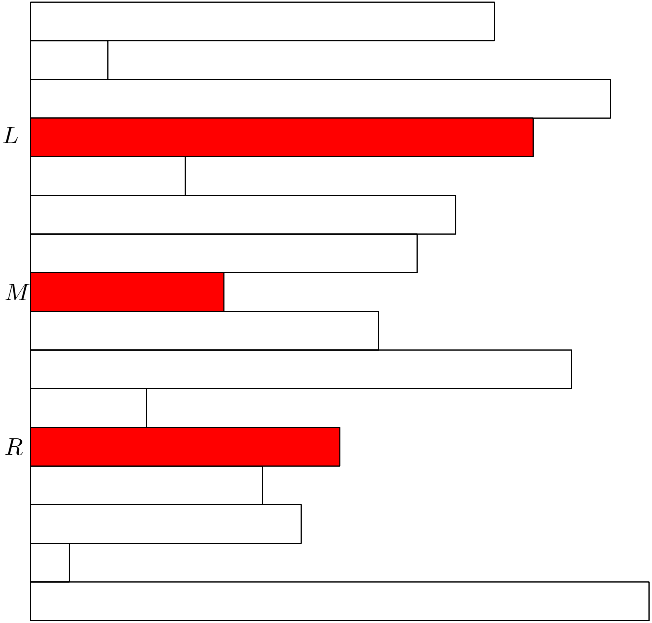
For the time being assume that we know how to answer RMQ queries on any array in $\mathcal{O}(1)$ time after linear preprocessing. Then we can compute any $\mathrm{lcp}(i, j)$ in $\mathcal{O}(1)$ time. Can this help us to speed up the binary searching?

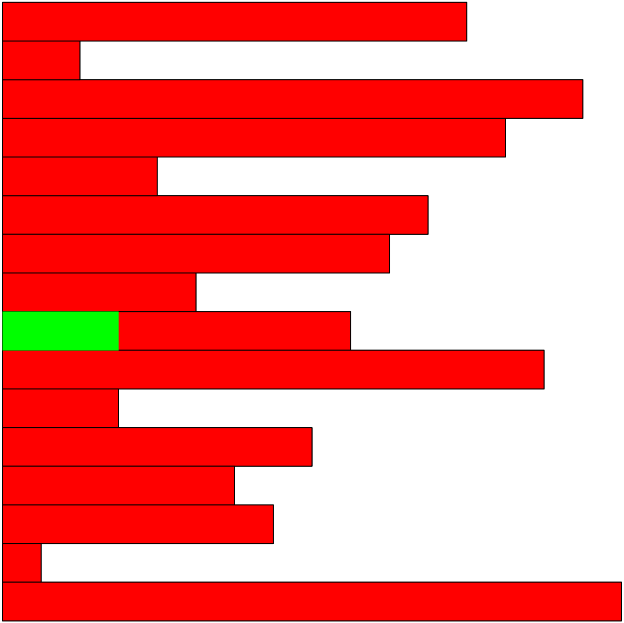
Now the question is whether we can do better. It seems that we are wasting lots of time comparing very similar blocks of texts again and again. Not cool!

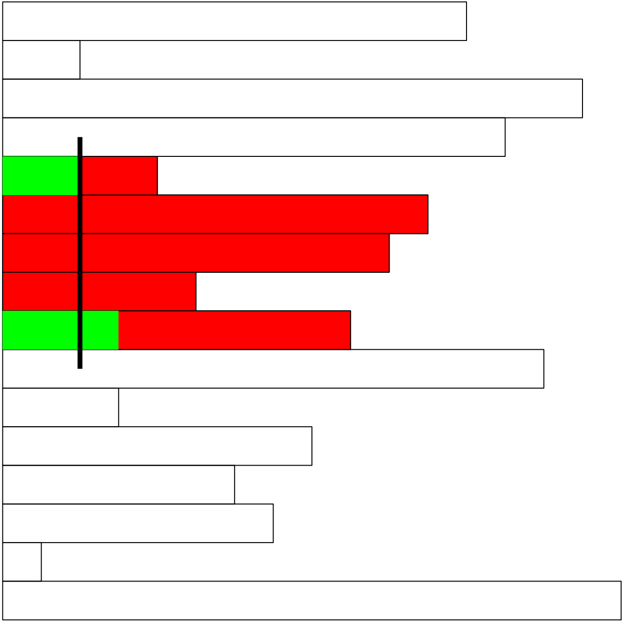
lcp again

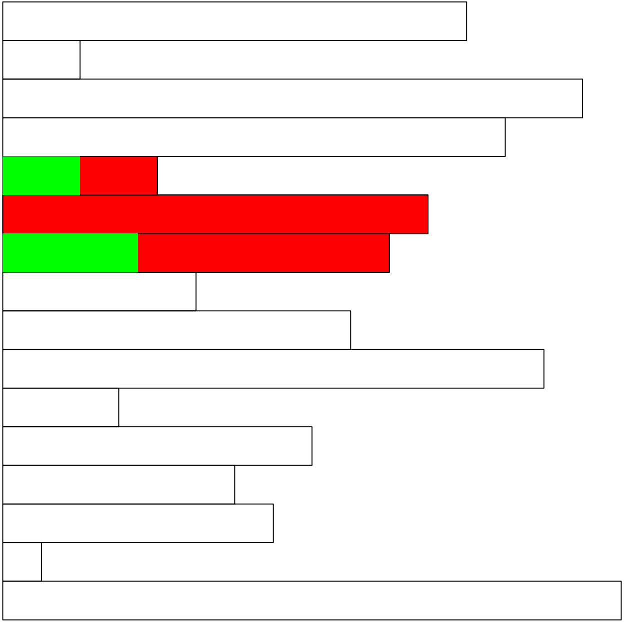
Recall that $\mathrm{lcp}(i, j)$ is the longest common prefix of the suffixes $w[i..n]$ and $w[j..n]$. We know how to compute all $\mathrm{lcp}[i]$, and we observed that computing $\mathrm{lcp}(i, j)$ reduces to the so-called Range Minimum Query on the $\mathrm{lcp}[i]$ array.

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.

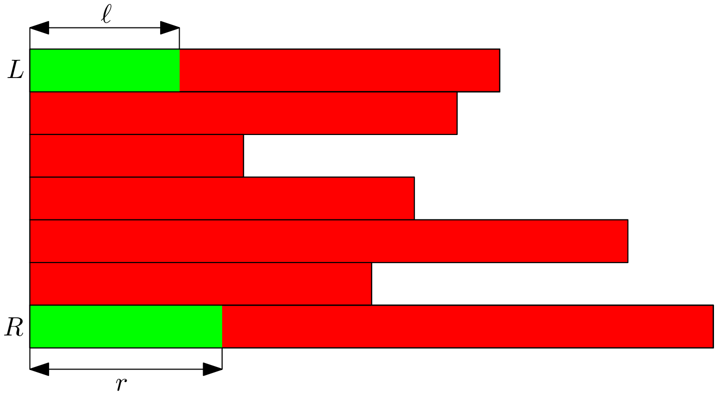
Let ℓ be the longest common prefix of $SA[L]$ and p, and r be the longest common prefix of $SA[R]$ and p. Assume that $\ell \leq r$, the situation is symmetric so the other case is very similar.

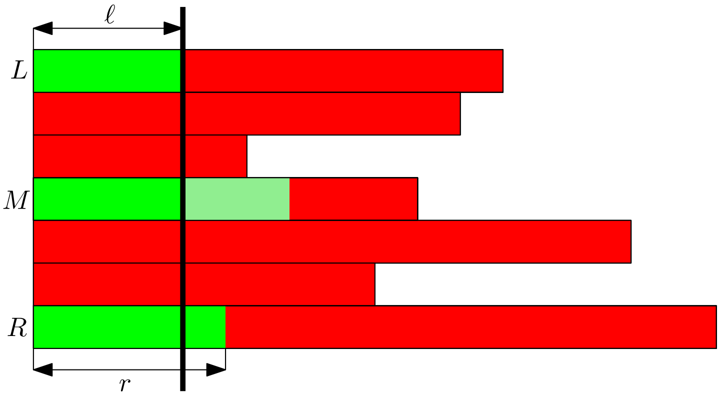
Invariant

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.

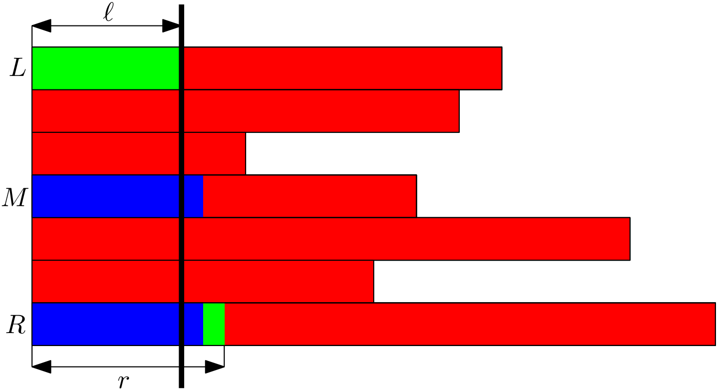
We choose $M \in (L, R)$. Of course we know that the longest common prefix of $SA[M]$ and p is at least as long as the minimum of the two known prefixes, but we can notice even more.

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.

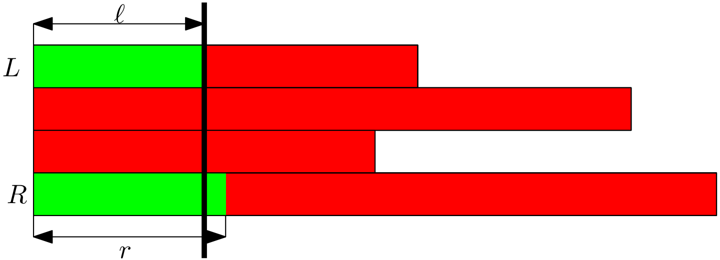

ℓ
L
R
r



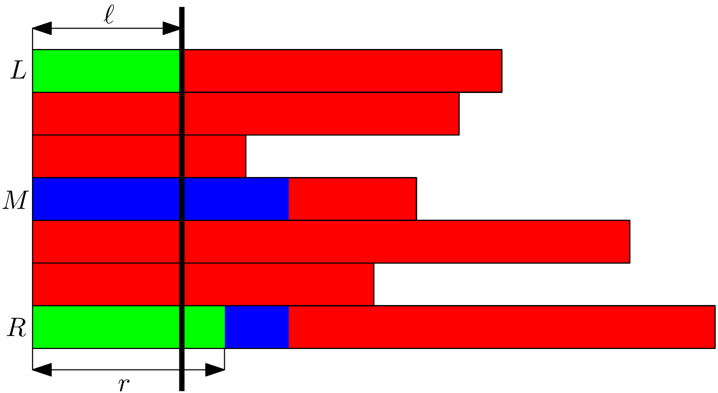
Look at $\mathrm{lcp}(SA[M], SA[R])$.



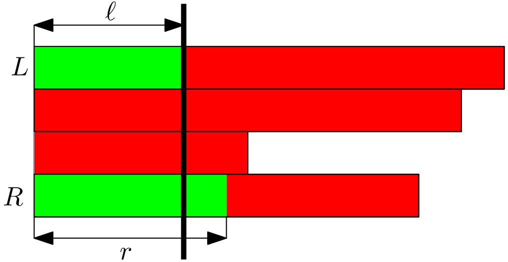
If $\mathrm{lcp}(SA[M], SA[R]) < r$, set $L = M$ and $\ell = \mathrm{lcp}(SA[M], SA[R])$.



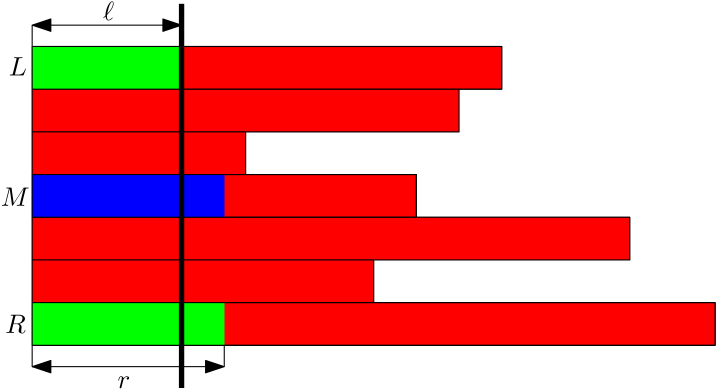
If $\mathrm{lcp}(SA[M], SA[R]) < r$, set $L = M$ and $\ell = \mathrm{lcp}(SA[M], SA[R])$.



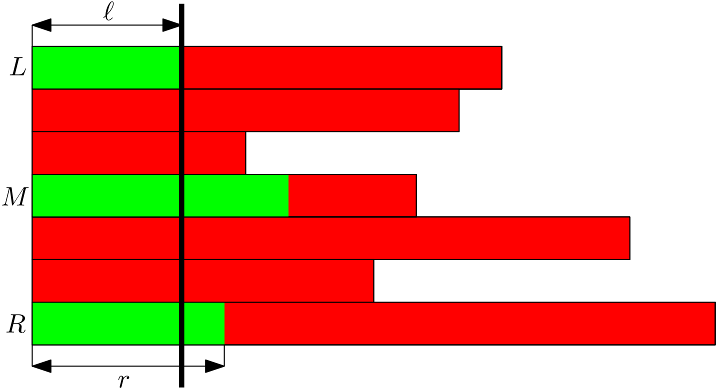
If $\operatorname{lcp}(SA[M], SA[R]) > r$, set $R = M$ and keep old ℓ and r.



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:

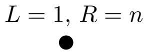
- the next character of $SA[M]$ is less than $p[k+1]$, then we set $L = M$ and $\ell = k$,
- the next character of $SA[M]$ is greater than $p[k+1]$, then we set $R = M$ and $r = k$.

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.

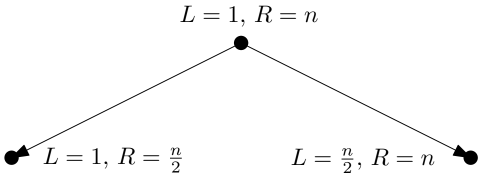
It follows that the sum of $\mathcal{O}(k - r)$ over all steps of the procedure is just $\mathcal{O}(m)$ in the worst possible case. We additionally spent $\mathcal{O}(1)$ time per step to look at $SA[M]$, hence the total complexity is $\mathcal{O}(m + \log n)$.

Recall that 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?



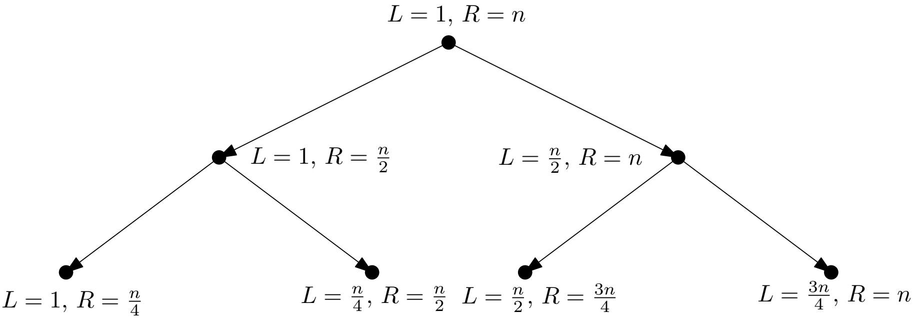
Each node of the recursion tree generates just two values $\mathrm{lcp}(SA[L], SA[M])$ and $\mathrm{lcp}(SA[M], SA[R])$ to be computed. Hence we have just $\mathcal{O}(n)$ values in total!

Recall that 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?



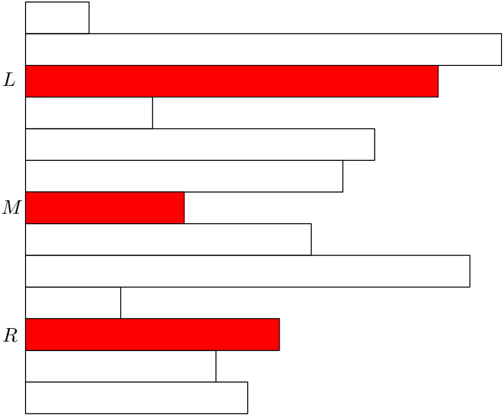
Each node of the recursion tree generates just two values $\mathrm{lcp}(SA[L], SA[M])$ and $\mathrm{lcp}(SA[M], SA[R])$ to be computed. Hence we have just $\mathcal{O}(n)$ values in total!

Recall that 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!

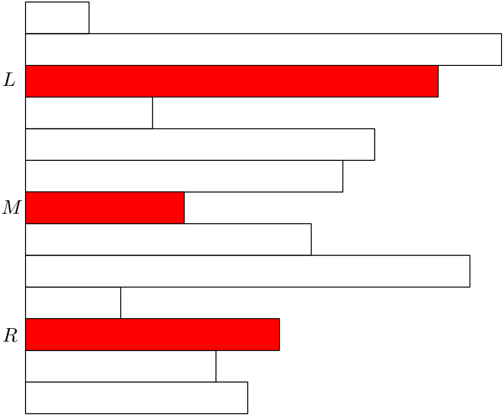
All those values can be actually computed in $\mathcal{O}(n)$ time in a bottom-top manner.



Lemma

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

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 yesterday 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 during the exercises). 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 yesterday 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 during the exercises). 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 yesterday 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 during the exercises). 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.

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

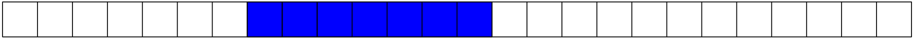
How? Well, $B_0[i] = A[i]$, and $B_{k+1}[i] = \min(B_k[i], B_k[i+2^k])$. Hence we can easily answer a query concerning a fragment of length that is a power of 2. But, unfortunately, not all numbers are powers of 2...

...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 $\min(B_k[i], B_k[j - 2^k + 1])$.

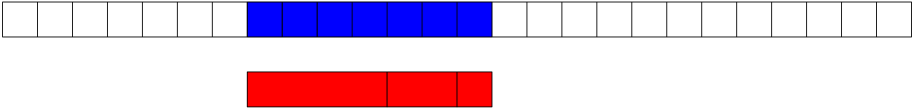
...or are they?



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?

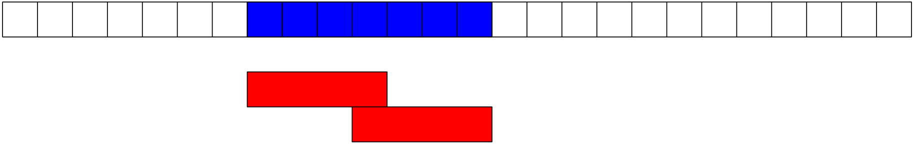


Any query can be split into at most $\log n$ power-of-two queries.

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?

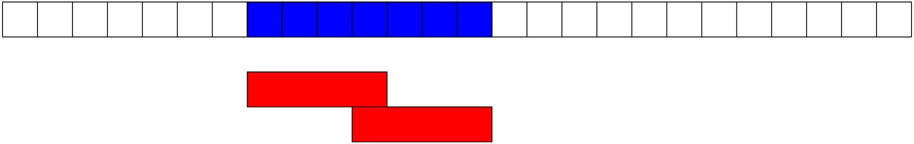


Any query can be covered with 2 power-of-two queries.

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: indirection. Chop the input array into blocks of length $b = \log n$.

Construct a new array A':

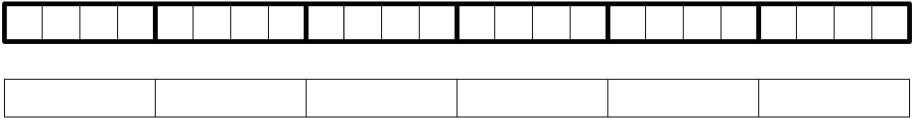
$$A'[i] = \min\{A[ib+1], A[ib+2], \ldots, A[(i+1)b]\}$$

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: indirection. Chop the input array into blocks of length $b = \log n$.



Construct a new array A':

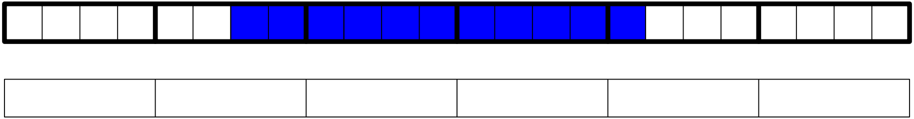
$$A'[i] = \min\{A[ib+1], A[ib+2], \ldots, A[(i+1)b]\}$$

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: indirection. Chop the input array into blocks of length $b = \log n$.



Construct a new array A':

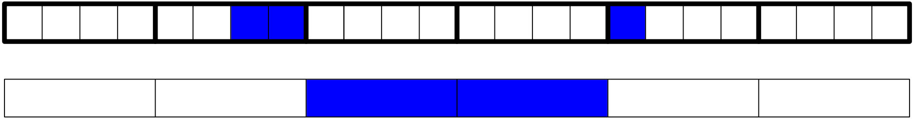
$$A'[i] = \min\{A[ib+1], A[ib+2], \ldots, A[(i+1)b]\}$$

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.

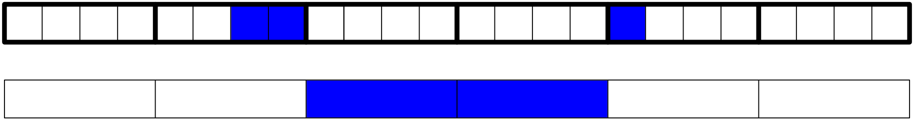
We apply another simple-yet-powerful technique: indirection. Chop the input array into blocks of length $b = \log n$.



Construct a new array A':

$$A'[i] = \min\{A[ib+1], A[ib+2], \ldots, A[(i+1)b]\}$$

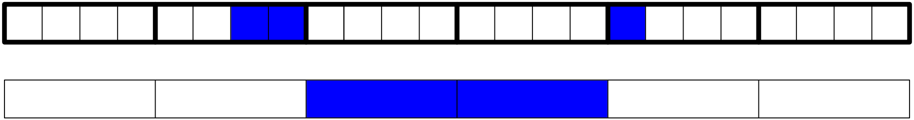
Build the previously described structure for A'.



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.

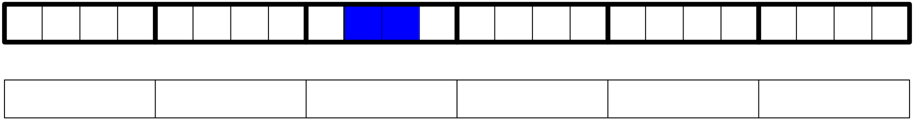
But the only case when we cannot answer a query in $\mathcal{O}(1)$ time is when the range is strictly inside a single block. Revert to the naive one-by-one computation!



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.

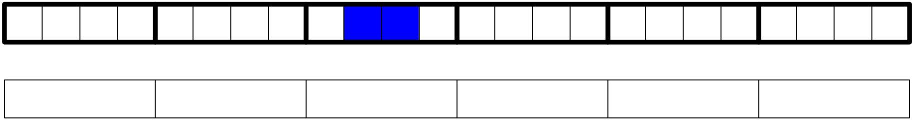
But the only case when we cannot answer a query in $\mathcal{O}(1)$ time is when the range is strictly inside a single block. Revert to the naive one-by-one computation!



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 by adding $2n + o(n)$ bits of space.

We "only" have to deal with the strictly-inside-a-block case.

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 by adding $2n + o(n)$ bits of space.

We “only” have to deal with the strictly-inside-a-block case.

We observe that the exact values of the elements don't matter that much, and we consider the so-called Cartesian tree of each block.

Cartesian tree

We choose the position p corresponding to the (leftmost) maximum to be the root. Then, we recursively define the Cartesian tree of the prefix before p, and attach it as the left child. Next, we recursively define the Cartesian tree of the suffix after p, and attach it as the right child.

What is the connection between the Cartesian tree and RMQ?

We observe that the exact values of the elements don't matter that much, and we consider the so-called Cartesian tree of each block.

Cartesian tree

We choose the position p corresponding to the (leftmost) maximum to be the root. Then, we recursively define the Cartesian tree of the prefix before p, and attach it as the left child. Next, we recursively define the Cartesian tree of the suffix after p, and attach it as the right child.

What is the connection between the Cartesian tree and RMQ?

Whole idea

When in doubt, use indirection... twice.

1. Partition the array into superblocks of length $s' = \log^{2+\epsilon} n$, preprocess the shorter array of length n/s' for RMQ.
2. Partition each superblock into blocks of length $s = \log n/(2+\delta)$, preprocess each short array of length s/s' for RMQ.
3. Finally, for every block store the shape of its Cartesian tree. The number of such trees is $\frac{1}{s+1}\binom{2s}{s} = 4^s/(\sqrt{\pi}s^{3/2})(1+\mathcal{O}(s^{-1}))$.

The overall space is

1. $n/s' \cdot \log n \cdot \log n = o(n)$
2. $n/s \cdot \log(s'/s) \cdot \log s' = o(n)$
3. $n/s \cdot \log(4^s/s^{3/2}) = n/s(2s - \mathcal{O}(\log s)) = 2n - \mathcal{O}(n \log\log n/\log n)$.

Do you see why we have applied indirection twice?

Whole idea

When in doubt, use indirection... twice.

1. Partition the array into superblocks of length $s' = \log^{2+\epsilon} n$, preprocess the shorter array of length n/s' for RMQ.
2. Partition each superblock into blocks of length $s = \log n/(2+\delta)$, preprocess each short array of length s/s' for RMQ.
3. Finally, for every block store the shape of its Cartesian tree. The number of such trees is $\frac{1}{s+1}\binom{2s}{s} = 4^s/(\sqrt{\pi}s^{3/2})(1+\mathcal{O}(s^{-1}))$.

The overall space is

1. $n/s' \cdot \log n \cdot \log n = o(n)$
2. $n/s \cdot \log(s'/s) \cdot \log s' = o(n)$
3. $n/s \cdot \log(4^s/s^{3/2}) = n/s(2s - \mathcal{O}(\log s)) = 2n - \mathcal{O}(n \log\log n/\log n)$.

Do you see why we have applied indirection twice?

Whole idea

When in doubt, use indirection... twice.

1. Partition the array into superblocks of length $s' = \log^{2+\epsilon} n$, preprocess the shorter array of length n/s' for RMQ.
2. Partition each superblock into blocks of length $s = \log n/(2+\delta)$, preprocess each short array of length s/s' for RMQ.
3. Finally, for every block store the shape of its Cartesian tree. The number of such trees is $\frac{1}{s+1}\binom{2s}{s} = 4^s/(\sqrt{\pi}s^{3/2})(1+\mathcal{O}(s^{-1}))$.

The overall space is

1. $n/s' \cdot \log n \cdot \log n = o(n)$
2. $n/s \cdot \log(s'/s) \cdot \log s' = o(n)$
3. $n/s \cdot \log(4^s/s^{3/2}) = n/s(2s - \mathcal{O}(\log s)) = 2n - \mathcal{O}(n \log\log n/\log n)$.

Do you see why we have applied indirection twice?

Whole idea

When in doubt, use indirection... twice.

1. Partition the array into superblocks of length $s' = \log^{2+\epsilon} n$, preprocess the shorter array of length n/s' for RMQ.
2. Partition each superblock into blocks of length $s = \log n/(2+\delta)$, preprocess each short array of length s/s' for RMQ.
3. Finally, for every block store the shape of its Cartesian tree. The number of such trees is $\frac{1}{s+1}\binom{2s}{s} = 4^s/(\sqrt{\pi}s^{3/2})(1+\mathcal{O}(s^{-1}))$.

The overall space is

1. $n/s' \cdot \log n \cdot \log n = o(n)$
2. $n/s \cdot \log(s'/s) \cdot \log s' = o(n)$
3. $n/s \cdot \log(4^s/s^{3/2}) = n/s(2s - \mathcal{O}(\log s)) = 2n - \mathcal{O}(n \log\log n/\log n)$.

Do you see why we have applied indirection twice?

Whole idea

When in doubt, use indirection... twice.

1. Partition the array into superblocks of length $s' = \log^{2+\epsilon} n$, preprocess the shorter array of length n/s' for RMQ.
2. Partition each superblock into blocks of length $s = \log n/(2+\delta)$, preprocess each short array of length s/s' for RMQ.
3. Finally, for every block store the shape of its Cartesian tree. The number of such trees is $\frac{1}{s+1}\binom{2s}{s} = 4^s/(\sqrt{\pi}s^{3/2})(1+\mathcal{O}(s^{-1}))$.

The overall space is

1. $n/s' \cdot \log n \cdot \log n = o(n)$
2. $n/s \cdot \log(s'/s) \cdot \log s' = o(n)$
3. $n/s \cdot \log(4^s/s^{3/2}) = n/s(2s - \mathcal{O}(\log s)) = 2n - \mathcal{O}(n \log\log n/\log n)$.

Do you see why we have applied indirection twice?

Whole idea

When in doubt, use indirection... twice.

1. Partition the array into superblocks of length $s' = \log^{2+\epsilon} n$, preprocess the shorter array of length n/s' for RMQ.
2. Partition each superblock into blocks of length $s = \log n/(2+\delta)$, preprocess each short array of length s/s' for RMQ.
3. Finally, for every block store the shape of its Cartesian tree. The number of such trees is $\frac{1}{s+1}\binom{2s}{s} = 4^s/(\sqrt{\pi}s^{3/2})(1+\mathcal{O}(s^{-1}))$.

The overall space is

1. $n/s' \cdot \log n \cdot \log n = o(n)$
2. $n/s \cdot \log(s'/s) \cdot \log s' = o(n)$
3. $n/s \cdot \log(4^s/s^{3/2}) = n/s(2s - \mathcal{O}(\log s)) = 2n - \mathcal{O}(n \log\log n/\log n)$.

Do you see why we have applied indirection twice?

Whole idea

When in doubt, use indirection... twice.

1. Partition the array into superblocks of length $s' = \log^{2+\epsilon} n$, preprocess the shorter array of length n/s' for RMQ.
2. Partition each superblock into blocks of length $s = \log n/(2+\delta)$, preprocess each short array of length s/s' for RMQ.
3. Finally, for every block store the shape of its Cartesian tree. The number of such trees is $\frac{1}{s+1}\binom{2s}{s} = 4^s/(\sqrt{\pi}s^{3/2})(1+\mathcal{O}(s^{-1}))$.

The overall space is

1. $n/s' \cdot \log n \cdot \log n = o(n)$
2. $n/s \cdot \log(s'/s) \cdot \log s' = o(n)$
3. $n/s \cdot \log(4^s/s^{3/2}) = n/s(2s - \mathcal{O}(\log s)) = 2n - \mathcal{O}(n \log\log n/\log n)$.

Do you see why we have applied indirection twice?

Whole idea

When in doubt, use indirection... twice.

1. Partition the array into superblocks of length $s' = \log^{2+\epsilon} n$, preprocess the shorter array of length n/s' for RMQ.
2. Partition each superblock into blocks of length $s = \log n/(2+\delta)$, preprocess each short array of length s/s' for RMQ.
3. Finally, for every block store the shape of its Cartesian tree. The number of such trees is $\frac{1}{s+1}\binom{2s}{s} = 4^s/(\sqrt{\pi}s^{3/2})(1+\mathcal{O}(s^{-1}))$.

The overall space is

1. $n/s' \cdot \log n \cdot \log n = o(n)$
2. $n/s \cdot \log(s'/s) \cdot \log s' = o(n)$
3. $n/s \cdot \log(4^s/s^{3/2}) = n/s(2s - \mathcal{O}(\log s)) = 2n - \mathcal{O}(n \log\log n/\log n)$.

Do you see why we have applied indirection twice?

Compressing lcp array

Suffix array clearly takes linear space: we only need to store the arrays SA, SA^{-1}, lcp, and the RMQ structure over lcp. Sounds great, but if we take a closer look, it might substantially exceed the size of the input. For example, if our string is binary, we need only n bits to represent it, and then the whole machinery adds $\mathcal{O}(n)$ words, which is $\mathcal{O}(n \log n)$ bits. Maybe we could do better?

Succinct RMQ

Given an array $A[1..n]$, we can built a structure consisting of $2n + o(n)$ bits, so that the position of a minimum of any fragment $A[i], A[i+1], \ldots, A[j]$ can be computed in $\mathcal{O}(1)$ time without accessing A.

See the problemset.

OK, but what about the lcp array?

Compressing lcp array

Suffix array clearly takes linear space: we only need to store the arrays SA, SA^{-1}, lcp, and the RMQ structure over lcp. Sounds great, but if we take a closer look, it might substantially exceed the size of the input. For example, if our string is binary, we need only n bits to represent it, and then the whole machinery adds $\mathcal{O}(n)$ words, which is $\mathcal{O}(n \log n)$ bits. Maybe we could do better?

Succinct RMQ

Given an array $A[1..n]$, we can built a structure consisting of $2n + o(n)$ bits, so that the position of a minimum of any fragment $A[i], A[i+1], \ldots, A[j]$ can be computed in $\mathcal{O}(1)$ time without accessing A.

See the problemset.

OK, but what about the lcp array?

Compressing lcp array

Suffix array clearly takes linear space: we only need to store the arrays SA, SA^{-1}, lcp, and the RMQ structure over lcp. Sounds great, but if we take a closer look, it might substantially exceed the size of the input. For example, if our string is binary, we need only n bits to represent it, and then the whole machinery adds $\mathcal{O}(n)$ words, which is $\mathcal{O}(n \log n)$ bits. Maybe we could do better?

Succinct RMQ

Given an array $A[1..n]$, we can built a structure consisting of $2n + o(n)$ bits, so that the position of a minimum of any fragment $A[i], A[i+1], \ldots, A[j]$ can be computed in $\mathcal{O}(1)$ time without accessing A.

See the problemset.

OK, but what about the lcp array?

Compressing lcp array

Suffix array clearly takes linear space: we only need to store the arrays SA, SA^{-1}, lcp, and the RMQ structure over lcp. Sounds great, but if we take a closer look, it might substantially exceed the size of the input. For example, if our string is binary, we need only n bits to represent it, and then the whole machinery adds $\mathcal{O}(n)$ words, which is $\mathcal{O}(n \log n)$ bits. Maybe we could do better?

Succinct RMQ

Given an array $A[1..n]$, we can built a structure consisting of $2n + o(n)$ bits, so that the position of a minimum of any fragment $A[i], A[i+1], \ldots, A[j]$ can be computed in $\mathcal{O}(1)$ time without accessing A.

See the problemset.

OK, but what about the lcp array?

Recall that we have a nice observation about lcp array:

$$\mathrm{lcp}[SA^{-1}[i]] - 1 \leq \mathrm{lcp}[SA^{-1}[i+1]]$$

Define $a(i) = \mathrm{lcp}[SA^{-1}[i]] + i - 1$. Then:

$$a(1) \leq a(2) \leq a(2) \leq \ldots \leq a(n-1) \leq a(n)$$

Furthermore, $a(i) \in [0, n]$, because the length of $w[i..n]$ is $n - i + 1$.

Recall that we have a nice observation about lcp array:

$$\mathrm{lcp}[SA^{-1}[i]] - 1 \leq \mathrm{lcp}[SA^{-1}[i+1]]$$

Define $a(i) = \mathrm{lcp}[SA^{-1}[i]] + i - 1$. Then:

$$a(1) \leq a(2) \leq a(2) \leq \ldots \leq a(n-1) \leq a(n)$$

Furthermore, $a(i) \in [0, n]$, because the length of $w[i..n]$ is $n - i + 1$.

Recall that we have a nice observation about lcp array:

$$\mathrm{lcp}[SA^{-1}[i]] - 1 \leq \mathrm{lcp}[SA^{-1}[i+1]]$$

Define $a(i) = \mathrm{lcp}[SA^{-1}[i]] + i - 1$. Then:

$$a(1) \leq a(2) \leq a(2) \leq \ldots \leq a(n-1) \leq a(n)$$

Furthermore, $a(i) \in [0, n]$, because the length of $w[i..n]$ is $n - i + 1$.

New (simpler) problem

How many bits of space do we need to store a nondecreasing sequence of numbers from $[0, n]$?

We store the differences between every two consecutive $a(i)$. The differences $a'(i) = a(i) - a(i-1)$ (where $a(0) = 0$) have the property that $a'(i) \geq 0$ and $\sum_i a'(i) = n$. So, it makes sense to store them as:

$$0^{a'(1)}10^{a'(2)}10^{a'(3)}\ldots 0^{a'(n-1)}10^{a'(n)}1$$

Extracting $a(i)$ reduces to counting zeroes before the i-th one.

We will show that a sequence of $2n$ bits can be stored using $2n + o(n)$ bits so that such operation can be performed in $\mathcal{O}(1)$ time.

New (simpler) problem

How many bits of space do we need to store a nondecreasing sequence of numbers from $[0, n]$?

We store the differences between every two consecutive $a(i)$. The differences $a'(i) = a(i) - a(i-1)$ (where $a(0) = 0$) have the property that $a'(i) \geq 0$ and $\sum_i a'(i) = n$. So, it makes sense to store them as:

$$0^{a'(1)}10^{a'(2)}10^{a'(3)}\ldots0^{a'(n-1)}10^{a'(n)}1$$

Extracting $a(i)$ reduces to counting zeroes before the i-th one.

We will show that a sequence of $2n$ bits can be stored using $2n + o(n)$ bits so that such operation can be performed in $\mathcal{O}(1)$ time.

New (simpler) problem

How many bits of space do we need to store a nondecreasing sequence of numbers from $[0, n]$?

We store the differences between every two consecutive $a(i)$. The differences $a'(i) = a(i) - a(i-1)$ (where $a(0) = 0$) have the property that $a'(i) \geq 0$ and $\sum_i a'(i) = n$. So, it makes sense to store them as:

$$0^{a'(1)}10^{a'(2)}10^{a'(3)}\ldots0^{a'(n-1)}10^{a'(n)}1$$

Extracting $a(i)$ reduces to counting zeroes before the i-th one.

We will show that a sequence of $2n$ bits can be stored using $2n + o(n)$ bits so that such operation can be performed in $\mathcal{O}(1)$ time.

New (simpler) problem

How many bits of space do we need to store a nondecreasing sequence of numbers from $[0, n]$?

We store the differences between every two consecutive $a(i)$. The differences $a'(i) = a(i) - a(i-1)$ (where $a(0) = 0$) have the property that $a'(i) \geq 0$ and $\sum_i a'(i) = n$. So, it makes sense to store them as:

$$0^{a'(1)}10^{a'(2)}10^{a'(3)} \ldots 0^{a'(n-1)}10^{a'(n)}1$$

Extracting $a(i)$ reduces to counting zeroes before the i-th one.

We will show that a sequence of $2n$ bits can be stored using $2n + o(n)$ bits so that such operation can be performed in $\mathcal{O}(1)$ time.

New (simpler) problem

How many bits of space do we need to store a nondecreasing sequence of numbers from $[0, n]$?

We store the differences between every two consecutive $a(i)$. The differences $a'(i) = a(i) - a(i-1)$ (where $a(0) = 0$) have the property that $a'(i) \geq 0$ and $\sum_i a'(i) = n$. So, it makes sense to store them as:

$$0^{a'(1)}10^{a'(2)}10^{a'(3)}\ldots 0^{a'(n-1)}10^{a'(n)}1$$

Extracting $a(i)$ reduces to counting zeroes before the i-th one.

We will show that a sequence of $2n$ bits can be stored using $2n + o(n)$ bits so that such operation can be performed in $\mathcal{O}(1)$ time.

Rank/select structure

Given a n-bit string, we want to add just $o(n)$ bits of additional information, which allow us to find in $\mathcal{O}(1)$ time:

- $\mathsf{rank}(i) =$ the number of ones at or before position i,
- $\mathsf{select}(i) =$ position of the i-th one.

Rank

Tabulation

Let $k = \frac{1}{2}\log n$. There are just $\sqrt{n}$ different binary strings of such size, so we can afford to precompute, for each such string, the answer for each possible rank query. The space required is just $\mathcal{O}(2^k k \log k) = o(n)$.

Now split the long string into fragments of length k. Store each such fragment in a single word, so that we can look-up the precomputed information quickly. Then, for each boundary between two fragments, store the cumulative rank.

Total space is $\frac{n}{k}\log n = \Theta(n)$ bits, too much.

Rank

Tabulation

Let $k = \frac{1}{2}\log n$. There are just $\sqrt{n}$ different binary strings of such size, so we can afford to precompute, for each such string, the answer for each possible rank query. The space required is just $\mathcal{O}(2^k k \log k) = o(n)$.

Now split the long string into fragments of length k. Store each such fragment in a single word, so that we can look-up the precomputed information quickly. Then, for each boundary between two fragments, store the cumulative rank.

Total space is $\frac{n}{k}\log n = \Theta(n)$ bits, too much.

Rank

Tabulation

Let $k = \frac{1}{2}\log n$. There are just $\sqrt{n}$ different binary strings of such size, so we can afford to precompute, for each such string, the answer for each possible rank query. The space required is just $\mathcal{O}(2^k k \log k) = o(n)$.

Now split the long string into fragments of length k. Store each such fragment in a single word, so that we can look-up the precomputed information quickly. Then, for each boundary between two fragments, store the cumulative rank.

Total space is $\frac{n}{k}\log n = \Theta(n)$ bits, too much.

But we can do better. Split the long string into fragments of length $\log^2 n$. For each boundary between two fragments, store the cumulative rank. This takes just $\mathcal{O}(\frac{n}{\log n})$ bits.

Then split each fragment into sub-fragments of size k. For each sub-fragment, store the cumulative rank **within** the fragment. This takes just $\mathcal{O}(\frac{n}{\log n} \log \log n)$ bits.

And we are done.

But we can do better. Split the long string into fragments of length $\log^2 n$. For each boundary between two fragments, store the cumulative rank. This takes just $\mathcal{O}(\frac{n}{\log n})$ bits.

Then split each fragment into sub-fragments of size k. For each sub-fragment, store the cumulative rank **within** the fragment. This takes just $\mathcal{O}(\frac{n}{\log n} \log \log n)$ bits.

And we are done.

But we can do better. Split the long string into fragments of length $\log^2 n$. For each boundary between two fragments, store the cumulative rank. This takes just $\mathcal{O}(\frac{n}{\log n})$ bits.

Then split each fragment into sub-fragments of size k. For each sub-fragment, store the cumulative rank **within** the fragment. This takes just $\mathcal{O}(\frac{n}{\log n} \log \log n)$ bits.

And we are done.

Select

Similar, but more complicated. Because we are looking for the i-th one, we split into fragments with the same number of ones.

Let $t_1 = \log n \log\log n$. We pick every t_1-th one and store its index in the whole string. This takes $\mathcal{O}(\frac{n}{t_1}\log n) = o(n)$ bits. Then, given a query, we divide it by t_1 to locate the desired fragment. Hence from now on we can focus on single fragments.

Select

Similar, but more complicated. Because we are looking for the i-th one, we split into fragments with the same number of ones.

Let $t_1 = \log n \log \log n$. We pick every t_1-th one and store its index in the whole string. This takes $\mathcal{O}(\frac{n}{t_1} \log n) = o(n)$ bits. Then, given a query, we divide it by t_1 to locate the desired fragment. Hence from now on we can focus on single fragments.

Let r be the total number of bits in a fragment.

$r > t_1^2$ things are sparse. There can be at most $\frac{n}{t_1^2}$ such fragments, and we can afford to store the index of each one in such fragment explicitly.

$r \leq t_1^2$ we cannot repeat the above simple trick, but things are not very bad, either. The fragment is short and **relative** indices can be stored.

More specifically, we repeat the reasoning, and split into subfragments containing $t_2 = (\log \log n)^2$ ones. For each one we pick, we store its relative index, which takes $\mathcal{O}(\frac{n}{t_2} \log \log n)$ bits in total. Then, again, we consider the total number bits r in a subfragment.

Let r be the total number of bits in a fragment.

$r > t_1^2$ things are sparse. There can be at most $\frac{n}{t_1^2}$ such fragments, and we can afford to store the index of each one in such fragment explicitly.

$r \leq t_1^2$ we cannot repeat the above simple trick, but things are not very bad, either. The fragment is short and **relative** indices can be stored.

More specifically, we repeat the reasoning, and split into subfragments containing $t_2 = (\log\log n)^2$ ones. For each one we pick, we store its relative index, which takes $\mathcal{O}(\frac{n}{t_2}\log\log n)$ bits in total. Then, again, we consider the total number bits r in a subfragment.

Let r be the total number of bits in a fragment.

$r > t_1^2$ things are sparse. There can be at most $\frac{n}{t_1^2}$ such fragments, and we can afford to store the index of each one in such fragment explicitly.

$r \leq t_1^2$ we cannot repeat the above simple trick, but things are not very bad, either. The fragment is short and **relative** indices can be stored.

More specifically, we repeat the reasoning, and split into subfragments containing $t_2 = (\log \log n)^2$ ones. For each one we pick, we store its relative index, which takes $\mathcal{O}(\frac{n}{t_2} \log \log n)$ bits in total. Then, again, we consider the total number bits r in a subfragment.

- $r > t_2^2$ things are sparse, and we store the relative index of each one. There are at most $\frac{n}{t_2^2}$ such subfragments, each contains t_2 ones, and relative indices take $\log\log n$ bits.
- $r \le t_2^2$ then $r \le \frac{1}{2}\log n$, and we use the tabulation trick.

Total space is $\mathcal{O}(\frac{n}{\log\log n})$ bits.

It often happens in this area that $o(n)$ means "something just a little bit below n", which is surely not what we would like if the result are to be of any relevance to the real world, but...

Pătraşcu 2008

For any constant c, rank/select can be implemented in $n + \mathcal{O}(\frac{n}{\log^c n})$ bits of space.

$r > t_2^2$ things are sparse, and we store the relative index of each one. There are at most $\frac{n}{t_2^2}$ such subfragments, each contains t_2 ones, and relative indices take $\log\log n$ bits.

$r \leq t_2^2$ then $r \leq \frac{1}{2}\log n$, and we use the tabulation trick.

Total space is $\mathcal{O}(\frac{n}{\log\log n})$ bits.

It often happens in this area that $o(n)$ means "something just a little bit below n", which is surely not what we would like if the result are to be of any relevance to the real world, but...

Pătraşcu 2008

For any constant c, rank/select can be implemented in $n + \mathcal{O}(\frac{n}{\log^c n})$ bits of space.

$r > t_2^2$ things are sparse, and we store the relative index of each one. There are at most $\frac{n}{t_2^2}$ such subfragments, each contains t_2 ones, and relative indices take $\log\log n$ bits.

$r \leq t_2^2$ then $r \leq \frac{1}{2}\log n$, and we use the tabulation trick.

Total space is $\mathcal{O}(\frac{n}{\log\log n})$ bits.

It often happens in this area that $o(n)$ means “something just a little bit below n”, which is surely not what we would like if the result are to be of any relevance to the real world, but...

Pătraşcu 2008

For any constant c, rank/select can be implemented in $n + \mathcal{O}(\frac{n}{\log^c n})$ bits of space.

$r > t_2^2$ things are sparse, and we store the relative index of each one. There are at most $\frac{n}{t_2^2}$ such subfragments, each contains t_2 ones, and relative indices take $\log\log n$ bits.

$r \leq t_2^2$ then $r \leq \frac{1}{2}\log n$, and we use the tabulation trick.

Total space is $\mathcal{O}(\frac{n}{\log\log n})$ bits.

It often happens in this area that $o(n)$ means "something just a little bit below n", which is surely not what we would like if the result are to be of any relevance to the real world, but...

Pătraşcu 2008

For any constant c, rank/select can be implemented in $n + \mathcal{O}(\frac{n}{\log^c n})$ bits of space.

Now we can store the lcp array and the RMQ structure in $4n + o(n)$ bits. But we still need to store SA, so we need $n \log n$ bits (we might also need to store SA^{-1}, which is another $n \log n$ bits). Now we will see how to decrease this bound!

Compressed suffix arrays

A text of length n over Σ can be stored in $n \log |\Sigma|$ bits. Now if Σ is small (think binary), $n \log n$ bits taken by the suffix array is way too much.

Compressed suffix arrays

Represent SA in $o(n \log n)$ bits of spaces, so that we can efficiently implement $\mathrm{lookup}(i)$ which returns $SA[i]$. (We don't care about extracting SA^{-1}.)

Grossi and Vitter 2000

For any constant $\epsilon > 0$, SA can be represented using just $(1 + \frac{1}{\epsilon}) n \log |\Sigma| + o(n \log |\Sigma|)$ bits, so that $\mathrm{lookup}(i)$ takes $\mathcal{O}(\log^{\epsilon} n)$.

Compressed suffix arrays

A text of length n over Σ can be stored in $n \log |\Sigma|$ bits. Now if Σ is small (think binary), $n \log n$ bits taken by the suffix array is way too much.

Compressed suffix arrays

Represent SA in $o(n \log n)$ bits of spaces, so that we can efficiently implement $\text{lookup}(i)$ which returns $SA[i]$. (We don't care about extracting SA^{-1}.)

Grossi and Vitter 2000

For any constant $\epsilon > 0$, SA can be represented using just $(1 + \frac{1}{\epsilon}) n \log |\Sigma| + o(n \log |\Sigma|)$ bits, so that $\text{lookup}(i)$ takes $\mathcal{O}(\log^{\epsilon} n)$.

Compressed suffix arrays

A text of length n over Σ can be stored in $n \log |\Sigma|$ bits. Now if Σ is small (think binary), $n \log n$ bits taken by the suffix array is way too much.

Compressed suffix arrays

Represent SA in $o(n \log n)$ bits of spaces, so that we can efficiently implement $\text{lookup}(i)$ which returns $SA[i]$. (We don't care about extracting SA^{-1}.)

Grossi and Vitter 2000

For any constant $\epsilon > 0$, SA can be represented using just $(1 + \frac{1}{\epsilon}) n \log |\Sigma| + o(n \log |\Sigma|)$ bits, so that $\text{lookup}(i)$ takes $\mathcal{O}(\log^{\epsilon} n)$.

Can we do even better?

The empirical entropy is the average number of bits per symbol needed to encode the text.

Entropy (or zeroth order empirical entropy)

$$H_0(T) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c}$$

where n_c is the number of occurrences of character c in T.

k-th order empirical entropy

$$H_k(T) = \frac{1}{n} \sum_{s \in \Sigma^k} |T_s| H_0(T_s)$$

where T_s is the concatenation of all characters in T following an occurrence of s.

It is known that Lempel-Ziv compression methods approach the k-th order empirical entropy.

Can we do even better?

The empirical entropy is the average number of bits per symbol needed to encode the text.

Entropy (or zeroth order empirical entropy)

$$H_0(T) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c}$$

where n_c is the number of occurrences of character c in T.

k-th order empirical entropy

$$H_k(T) = \frac{1}{n} \sum_{s \in \Sigma^k} |T_s| H_0(T_s)$$

where T_s is the concatenation of all characters in T following an occurrence of s.

It is known that Lempel-Ziv compression methods approach the k-th order empirical entropy.

Can we do even better?

The empirical entropy is the average number of bits per symbol needed to encode the text.

Entropy (or zeroth order empirical entropy)

$$H_0(T) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c}$$

where n_c is the number of occurrences of character c in T.

k-th order empirical entropy

$$H_k(T) = \frac{1}{n} \sum_{s \in \Sigma^k} |T_s| H_0(T_s)$$

where T_s is the concatenation of all characters in T following an occurrence of s.

It is known that Lempel-Ziv compression methods approach the k-th order empirical entropy.

Can we do even better?

The empirical entropy is the average number of bits per symbol needed to encode the text.

Entropy (or zeroth order empirical entropy)

$$H_0(T) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c}$$

where n_c is the number of occurrences of character c in T.

k-th order empirical entropy

$$H_k(T) = \frac{1}{n} \sum_{s \in \Sigma^k} |T_s| H_0(T_s)$$

where T_s is the concatenation of all characters in T following an occurrence of s.

It is known that Lempel-Ziv compression methods approach the k-th order empirical entropy.

Can we do even better?

Now we would like to represent *SA* in space proportional to the *k*-th order empirical entropy of the text.

Sadakane 2003

For any constant $\epsilon, \epsilon' > 0$, *SA* can be represented using $H_0(T)n\frac{1+\epsilon'}{\epsilon} + n(2\log(1 + H_0(T)) + 3) + o(n)$ bits, so that $\text{lookup}(i)$ takes $\mathcal{O}(\frac{1}{\epsilon\epsilon'}\log^{\epsilon} n)$ time, assuming $|\Sigma| = \text{polylog}(n)$.

Grossi, Gupta, Vitter 2003

SA can be represented using $H_k(T)n + \mathcal{O}(n\log|\Sigma|\frac{\log\log n}{\log n})$ bits, so that $\text{lookup}(i)$ takes $\mathcal{O}(\log^{1+\epsilon} n)$ time.

These bounds are painful to look at, so we will ignore them.

Can we do even better?

Now we would like to represent *SA* in space proportional to the k-th order empirical entropy of the text.

Sadakane 2003

For any constant $\epsilon, \epsilon' > 0$, *SA* can be represented using $H_0(T)n\frac{1+\epsilon'}{\epsilon} + n(2\log(1 + H_0(T)) + 3) + o(n)$ bits, so that $\text{lookup}(i)$ takes $\mathcal{O}(\frac{1}{\epsilon\epsilon'}\log^{\epsilon} n)$ time, assuming $|\Sigma| = \text{polylog}(n)$.

Grossi, Gupta, Vitter 2003

SA can be represented using $H_k(T)n + \mathcal{O}(n \log|\Sigma|\frac{\log\log n}{\log n})$ bits, so that $\text{lookup}(i)$ takes $\mathcal{O}(\log^{1+\epsilon} n)$ time.

These bounds are painful to look at, so we will ignore them.

Can we do even better?

Now we would like to represent *SA* in space proportional to the k-th order empirical entropy of the text.

Sadakane 2003

For any constant $\epsilon, \epsilon' > 0$, *SA* can be represented using $H_0(T)n^{\frac{1+\epsilon'}{\epsilon}} + n(2\log(1 + H_0(T)) + 3) + o(n)$ bits, so that lookup(i) takes $\mathcal{O}(\frac{1}{\epsilon\epsilon'}\log^{\epsilon} n)$ time, assuming $|\Sigma| =$ polylog(n).

Grossi, Gupta, Vitter 2003

SA can be represented using $H_k(T)n + \mathcal{O}(n\log|\Sigma|\frac{\log\log n}{\log n})$ bits, so that lookup(i) takes $\mathcal{O}(\log^{1+\epsilon} n)$ time.

These bounds are painful to look at, so we will ignore them.

Can we do even better?

Now we would like to represent *SA* in space proportional to the k-th order empirical entropy of the text.

Sadakane 2003

For any constant $\epsilon, \epsilon' > 0$, *SA* can be represented using $H_0(T)n\frac{1+\epsilon'}{\epsilon} + n(2\log(1 + H_0(T)) + 3) + o(n)$ bits, so that lookup(i) takes $\mathcal{O}(\frac{1}{\epsilon\epsilon'}\log^{\epsilon} n)$ time, assuming $|\Sigma| = \text{polylog}(n)$.

Grossi, Gupta, Vitter 2003

SA can be represented using $H_k(T)n + \mathcal{O}(n \log|\Sigma| \frac{\log\log n}{\log n})$ bits, so that lookup(i) takes $\mathcal{O}(\log^{1+\epsilon} n)$ time.

These bounds are painful to look at, so we will ignore them.

Grossi and Vitter

We will assume $|\Sigma| = 2$.

SA can be represented in $\frac{1}{2}n \log \log n + 6n + \mathcal{O}(\frac{n}{\log \log n})$ bits, so that lookup(i) takes $\mathcal{O}(\log \log n)$ time.

Questions?