Implementation and comparison of LZ-Index and FM-Index algorithms

Mateusz Hurkała

Bachelor thesis
Tutor: D.Sc. Krzysztof Turowski

Jagiellonian University
Department of Theoretical Computer Science
Kraków 2024

Contents

Chapter 1

Introduction

1.1 String-searching

1.1.1 Definitions

Let us first introduce basic concepts, used through this work. Let $\mathcal{A}$ denote an alphabet, that is the set of characters. A text, denoted as t, is the sequence of characters from the alphabet $\mathcal{A}$, we denote reversed text t as t^r. A pattern s is a text that we will try to find in given text t.

Definition 1. *Substring $t[i : j+1]$ is a part of the text $t = a_1...a_n$ such that $t[i : j+1] = a_i, a_{i+1}...a_{j-1}, a_j$.*

Definition 2. *Suffix $t[i :]$ is a substring of text $t = a_1...a_n$ such that $t[i :] = a_i...a_n$*

Definition 3. *Prefix $t[: j+1]$ is a substring of text $t = a_1...a_n$, such that $t[: j+1] = a_1...a_j$.*

Definition 4. *Compression of text t is a function $F : \mathcal{A}^{\mathbb{N}} \mapsto X^{\mathbb{N}}$, that maps text to any object used for compression. We denote length of the result of compression as w.*

Definition 5. *Compression rate is the ratio of the bit representation of uncompressed text and compressed text. That is $\frac{x}{y}$, where x is the size of uncompressed text in bits and y is length of bit representation of its compression.*

1.1.2 Problem definitions

String-searching is well known problem which can also be found in everyday life.

Problem 1 (String-searching)**.** *[Bae89] The string searching, sometimes also called string matching problem, consists of finding all occurrences (or the first occurrence) of a pattern s in a text t.*

Problem 2 (Multiple string-searching)**.** *[CL16] Given a finite set of k pattern strings $S = \{s_1, s_2, , ..., s_k\}$ and a text t. The problem is to find all the text positions where the s_i occurs in t. More precisely the problem is to compute the set $\{j : \exists_i \; s_i = t_j t_{j+1} ... t_{j+|Pi|-1}\}$.*

Problem 3 (String-searching with mismatches)**.** *[NR13] Given a text $t = a_1a_2...a_n$, and a pattern $s = s_1s_2...s_m$, return, index i, $(1 \leq i \leq n-m+1)$, such that strings $t[i : i+m+1]$, s differ at the minimal number of positions.*

Problem 4 (Wildcard matching)**.** *[CC07] Let text $t = t_1...t_n$ and pattern $s = s_1...s_m$. The pattern s is said to occur at location i in t if, for every position j in the pattern, either $s_j = t_{i+j}$ or at least one of s_j and t_{i+j} is the wildcard symbol.*

Through this work we focus only on the string-searching and the multiple string-searching problems. The most we will focus on the multiple string-searching problem due to the fact that we can save much more time solving that version compared to naive algorithms and it has more usages in practice.

1.2 Usage of string-searching in practice

String-searching is used in many areas of IT and the easiest algorithms to solve those problems are already implemented in some programming languages by default [Kap+14] [Gus97]. String-searching can be used in:

- Spell Checker
- Spam Filter
- Intrusion Detection System Model
- Search Engine Module
- Plagiarism Detection System
- DNA Sequencing Module
- Digital Forensic Results
- Information Retrieval Modal
- Databases
- Search tools, such as grep
- Educational CD-ROM applications
- Internet browsers and crawlers

As can be seen, a lot of branches requires to solve that problem and many of them are commonly used in the daily life.

1.3 Simple algorithms for string-searching

1.3.1 Naive search

Most natural approach to solve string-searching is naive search where for each index i in text t we compare a substring $t[i : i + m]$ with the pattern. This algorithm can be written as follows:

```
# t - text, s - pattern, n - length of t, m - length of s
def naive_search(t, s, n, m):
    for i in range(0, n-m):
        equals_on_index = True
        for j in range(0,m):
            if t[i+j] != s[j]:
                equals_on_index = False
                break
        if equals_on_index:
            return True
    return False
```

It is very simple algorithm, but in the worst case it can be very slow. For example $t = a^n$, $s = a^{m-1}b$ requires $(n - m) \cdot m$ operations. It is easy to observe that the worst time complexity will be $\mathcal{O}(n \cdot m)$. For version with multiple patterns we have to run this algorithm for each pattern independently, which results in $\mathcal{O}(n \cdot \sum m_i)$ running time.

However, if the patterns are generated randomly and the size of the alphabet is greater than 1, this algorithm is quite fast. We observe that with such assumptions the expected number of comparisons in inner loop will be constant and for version with q patterns we obtain $\mathcal{O}(q \cdot n)$ running time.

Table 1.1: Comparison of simple algorithms

Algorithm	Average case	Worst case	Additional space
Naive algorithm	$\mathcal{O}(n)$	$\mathcal{O}(n \cdot m)$	$\mathcal{O}(1)$
Rabin-Karp [KR87]	$\mathcal{O}(n)$	$\mathcal{O}(n \cdot m)$	$\mathcal{O}(1)$
Knuth-Morris-Pratt [KMP77]	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(m)$
Two-way algorithm [CP91]	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log m)$

If the average complexity of a naive algorithm is not enough in specific case there are some algorithms mentioned in table 1.1, which for single pattern achieves optimal running time.

1.4 Suffix array, BWT and Wavelet tree

1.4.1 Suffix Array

Suffix array introduced in [UG90] is a data structure which describes the suffixes of the text. It stores all the suffixes of text sorted in lexicographic order, but there is one trick, it stores only the beginning index of each suffix.

Definition 6. *[Cor+22, p. 986] Given a text $t = a_1, ..., a_n$. The suffix array SA of t is defined such that if $SA[i] = j$, then $t[j]$ is the i-th suffix of t in lexicographic order. That is, the i-th suffix of t in lexicographic order is $t[SA[i] :]$.*

We add character \$ at the end of text which will work as an empty suffix and that character will also be less than any other character.

Visualization of suffix array

Index	Suffix
0	$ananas\$$
1	$nanas\$$
2	$anas\$$
3	$nas\$$
4	$as\$$
5	$s\$$
6	\$

$\Longrightarrow$

SA-Index	Original index	Suffix
0	6	\$
1	0	$ananas\$$
2	2	$anas\$$
3	4	$as\$$
4	1	$nanas\$$
5	3	$nas\$$
6	5	$s\$$

If we want to store each suffix as strings, it will require at least $\mathcal{O}(n^2)$ time and space, which is not optimal. The suffix array can be computed by many methods, such as:

Table 1.2: Algorithms for computing suffix array

Algorithm	Time complexity	Additional space
Naive algorithm	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(n^2)$
KMR algorithm [KMR72]	$\mathcal{O}(n \log^2 n)$	$\mathcal{O}(n \log n)$
KMR [KMR72] + radix sort [Dav92]	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
SA-IS algorithm [NZC09]	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Kärkkäinen-Sanders algorithm [KS03]	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Larsson-Sadakane algorithm [LS07]	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
Linear suffix tree algorithm [Far97]	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Suffix array is used in many text algorithms. We will use this structure later for computing FM-index, but if we have suffix array for example we can solve string-matching for many patterns faster than naive algorithms. Note that the suffix array can be computed in $\mathcal{O}(n)$ running time using algorithms mentioned in table 1.2.

Let us consider the text t of length n, it's suffix array SA and patterns $s_1, s_2....$ To solve string-matching using suffix array we will make simple observation which will allow us to use binary search.

Observation 1. *Pattern s exists in text t if and only if there is a suffix of t that have a prefix equals to s.*

Having that observation and a suffix array we can use binary search over the suffix array to find such suffix. Due to the definition of suffix array, all suffixes in it are sorted so that we can use binary search.

```python
# t - text, SA - suffix array, n - length of t, s - pattern
# m - length of s
def pattern_exists(t, SA, n, s, m)
    l = 0
    r = n - 1
    while l > r:
        mid = (l+r)//2
        string_to_compare = ''.join(
            t[i] for i in range(SA[mid], min(n, SA[mid] + m + 1)))
        if string_to_compare < s:
            l = mid + 1
        else:
            r = mid
    if  (n - SA[l] < m) or (t[SA[l] : SA[l] + m] != s):
        return False
    return True
```

As we can see, the binary search will execute string comparison of length at most m, so the time complexity of this function is $\mathcal{O}(\log(n) \cdot m)$, which after the summation for all strings gives us $\mathcal{O}(\log(n) \cdot \sum m_i)$, which is not optimal because the input size is $\sum m_i$. In that analysis we are skipping the time needed for calculating the suffix array, which can be done using $\mathcal{O}(n)$ time. Comparison of suffix and pattern can be stopped on the first index that are different, so we can save some time comparing characters one by one without materializing the suffix, but the worst complexity will be the same.

1.4.2 Burrows-Wheeler transform (BWT)

The Burrows-Wheeler transform (BWT) is a transformation that permutes the characters of a string in a specific order. The easiest way to show how BWT work is to use an example.

Let us consider text $t = baabb$. We can mark the end of the text with $\$$ to make it reversible, so $t = baabb\$$. Now let us construct a matrix of all the circular shifts of the text (circular shift on text is an operation that takes the first character from text and place it at the end of the text).

$$\begin{bmatrix} baabb\$ \\ aabb\$b \\ abb\$ba \\ bb\$baa \\ b\$baab \\ \$baabb \end{bmatrix}$$

Now we sort the texts in the matrix in lexicographic order. The matrix after sort will look like this:

$$\begin{bmatrix} \$baabb \\ aabb\$b \\ abb\$ba \\ baabb\$ \\ bb\$baa \\ b\$baab \end{bmatrix}$$

The output of BWT is the last column of this matrix, that is, $BWT(t) = bba\$ab$.

Definition 7. *$BWT(t)$ is the last column of the matrix of circular shifts, sorted lexicographically.*

Such definition immediately provide an algorithm for computing $BWT(t)$, which is, get all circular shifts of t, sort them and get the last column of matrix. Unfortunately, a direct approach is very slow, because sorting such a matrix in the worst case requires $\mathcal{O}(n^2 \log n)$ time, but it can be done much faster. This algorithm works similar to the naive algorithm for computing the suffix array because, the matrix after sort is like a suffix array (but it does not include the indexes), in detail the characters in the first column are the letters on indexes from suffix array. All the texts in this matrix are circular shifts, so the character in the last column is a previous character in text of the character in the first column. This follows to the new observation.

Observation 2. *$BWT(t)$ on index i is the same as $t[SA[i]-1]$ or \$ if $SA[i] = 0$, and at the first index of $BWT(t)$ is last character of t.*

This observation provides a fast algorithm for computing $BWT(t)$, that requires a precomputed suffix array.

```
def BWT(SA, t, n):
    bwt = [t[n - 1]]
    for i in range(0, n):
        if SA[i] == 0:
            bwt.append('$')
        else:
            bwt.append(t[SA[i] - 1])
    return ''.join(bwt)
```

It can be proved that this algorithm requires only $\mathcal{O}(n)$ time, which means that it is optimal for computing $BWT(t)$.

1.4.3 Wavelet tree

Wavelet tree is a data structure for performing a certain set of operations on ranges for a given text. In particular, the required operations are:

- `rank(i, j, x)` – get the number of occurrences of character x in $t[i : j + 1]$.
- `select(i, j, k, x)` – get the k-th occurrence of character x in $t[i : j + 1]$.
- `quantile(i, j, k)` – get the k-th smallest character in $t[i : j + 1]$.
- `range_count(i, j, x, y)` – get the number of positions $k \in [i, j]$ such that $x \leq t[k] \leq y$.

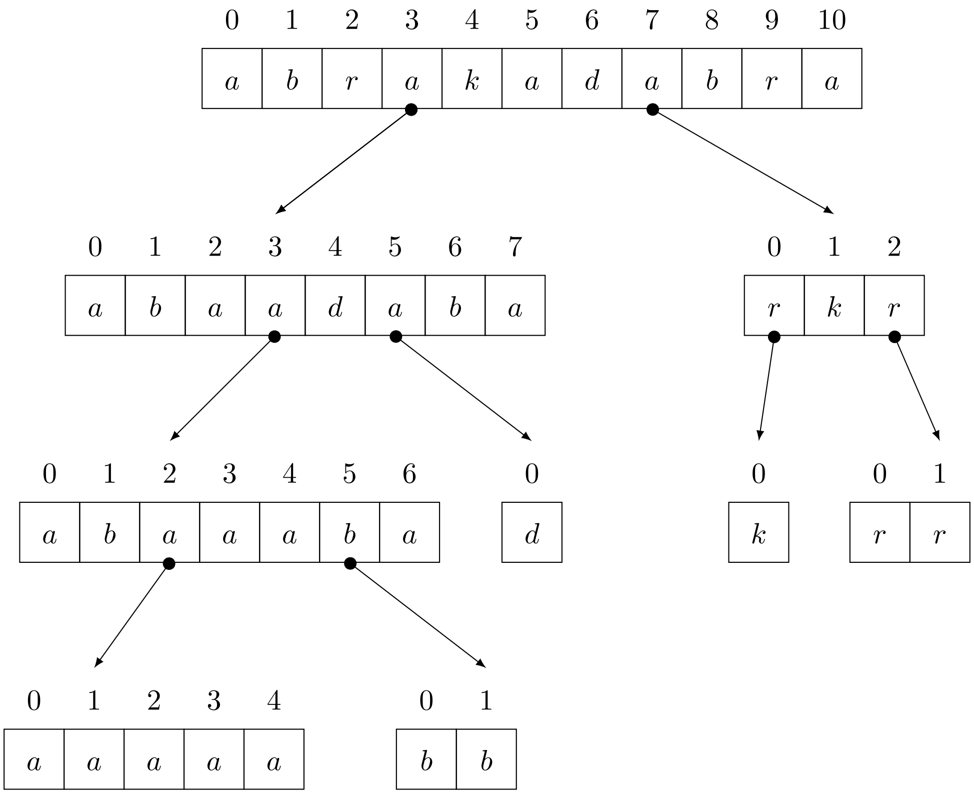
The wavelet tree can be constructed in $\mathcal{O}(n \cdot \log(|\mathcal{A})|)$ time and each of the queries mentioned above can be answered in $\mathcal{O}(\log(|\mathcal{A}|))$ time. The only weakness of wavelet tree is its space complexity, equal to $\mathcal{O}(n \cdot \log(|\mathcal{A}|))$.

Idea

The idea of Wavelet Tree is quite simple. We will represent Wavelet Tree as a binary tree. In each node we store the alphabet of t, minimal and maximal character in alphabet, prefix sum array and the information about each character and index to which division does it belong.

For each node we sort an alphabet and divide it in half denoted as $\mathcal{A}_1, \mathcal{A}_2$ and create texts t_1, t_2 where t_1 (respectively, t_2) is a copy of t with all characters for alphabet $\mathcal{A}_2$ (respectively, $\mathcal{A}_1$) removed, but the order of characters exactly preserved as in t. Next we recursively create left node with arguments $t_1, \mathcal{A}_1$ and right node with arguments $t_2, \mathcal{A}_2$, until we reach the base case, when the alphabet $\mathcal{A}$ contains only one element. Additionally, in each node we store a prefix sum array where the character contributes to the prefix sum as 0 where character is in $\mathcal{A}_1$ and as 1 if character is in $\mathcal{A}_2$.

Example of Wavelet Tree for $t = abrakadabra$



To save time we can sort the alphabet only once in the root node, split it and pass it as an argument for children nodes, which will result in total $\mathcal{O}(|\mathcal{A}| \log(|\mathcal{A}|))$ operations for sorting.

A construction of Wavelet Tree can be implemented in the following way:

```python
class WaveletTree:
  def __init__(self, t, n, sorted_alphabet = None):
    t = t[1:]
    if sorted_alphabet is not None:
      self.alphabet = set(sorted_alphabet)
    else:
      self.alphabet = set(t)
      sorted_alphabet = sorted(list(self.alphabet))
    self.n = n
    self.smallest = sorted_alphabet[0]
    self.largest = sorted_alphabet[-1]
    if len(sorted_alphabet) == 1:
      self.leaf = True
      return
    self.leaf = False
    left_alphabet = sorted_alphabet[:(len(sorted_alphabet) + 1)//2]
    right_alphabet = sorted_alphabet[(len(sorted_alphabet) + 1)//2:]
    self.zero_indexed = set(left_alphabet)
    self.one_indexed = set(right_alphabet)
    value_array = [1 if c in self.one_indexed else 0 for c in t ]
    self.prefix_sum = [0]
    for i in range(n):
      self.prefix_sum.append(self.prefix_sum[i] + value_array[i])
    self.left_indices = [0]
    self.right_indices = [0]
    for i in range(n):
      if t[i] in self.zero_indexed:
        self.left_indices.append(i+1)
      else:
        self.right_indices.append(i+1)
    left_t = ['#'] + [c for c in t if c in self.zero_indexed]
    right_t = ['#'] + [c for c in t if c in self.one_indexed]
    self.left = WaveletTree(left_t, len(left_t) - 1, left_alphabet)
    self.right = WaveletTree(right_t, len(right_t) - 1, right_alphabet)
```

It can be seen that the creation of Wavelet Tree requires $\mathcal{O}(n \log(|\mathcal{A}|))$ time and space. We can observe that for each level of nodes the total time and space is the same, and it is equal to $\mathcal{O}(n)$ also the height of the tree is $\mathcal{O}(\log(|\mathcal{A}|))$. The creation complexity can be compared to the complexity of Merge Sort [Cor+22, p. 34-44] where we use similar approach. Also note that $|\mathcal{A}|$ is less or equal to n, so the time used for the sorting at the root node will not be greater than $\mathcal{O}(n \log(|\mathcal{A}|))$.

Using all the information stored in Wavelet Tree we can now proceed to answer the queries.

Rank query

`rank(i, j, x)` query can be answered using recursion and modifying the range as we go deeper into the tree to the leaf, it can be described in the following way:

1. If x is not in the alphabet or range $[i, j]$ is empty return 0.
2. If the node is a leaf, then return $r - l + 1$.
3. Modify i and j, so they correspond to the correct positions in a recursive call.
4. Recursive call on proper child and range.

The range modification may be implemented in the following way:

```
def _left_tree_range(self, l, r):
  return l - self.prefix_sum[l-1], r - self.prefix_sum[r]

def _right_tree_range(self, l, r):
  return (self.prefix_sum[l-1] + 1, self.prefix_sum[r])
```

The correctness of each query follows explicitly from the correctness of range mapping and the trivially true base case.

For the right node we observe that the prefix sum correspond to the number of elements in right tree at given index, so new indexes in right tree are just values of prefix sum. In the other case the indices in left node are the complement for right node, so it will be original index minus the prefix sum value. Using such functions, the `rank(i, j, x)` query can be implemented in the following way:

```
def rank(self, c, l, r):
  if c not in self.alphabet or l > r or l > self.n or r < 1:
    return 0
  if self.leaf:
    return r-l+1
  if c in self.zero_indexed:
    new_l, new_r =  self._left_tree_range(l, r)
    return self.left.rank(c, new_l, new_r)
  new_l, new_r =  self._right_tree_range(l, r)
  return self.right.rank(c, new_l, new_r)
```

Note that it is easy to modify `rank(c, l, r)` function to return all indices. The only difference is that we return range $[l, r]$ in the base case and map all indices from the recursion call to current node indices.

Select query

Answering the `select(i, j, k, x)` query uses the same approach as the `rank(i, j, x)` query but returns another information in the base case where we just return an index and as we leave the recursion call we map it to an index in currently considered text inside a node. It may be implemented as follows:

```
def select(self, c, k, l, r):
  if c not in self.alphabet or l > r or l > self.n or r < 1 :
    return None
  if self.leaf:
    return k+l-1 if k <= r-l+1 else None
  if c in self.zero_indexed:
    new_l, new_r =  self._left_tree_range(l, r)
    rec_result = self.left.select(c, k, new_l, new_r)
    return (self.left_indices[rec_result] if rec_result is not None
      else None)
  new_l, new_r =  self._right_tree_range(l, r)
  rec_result = self.right.select(c, k, new_l, new_r)
  return (self.right_indices[rec_result] if rec_result is not None
      else None)
```

Quantile query

The `quantile(i, j ,k)` will also use similar approach but in each step we will count the number of characters in the left node in the corresponding range and recursively call the function according to the k and number of characters. As can be seen if number of characters in the left node in range $[i, j]$ is less than k we will call recursively with arguments $mapped(i)$, $mapped(j)$ and $k - num$, due to that each character in left node are smaller than in the right node, and now we are searching for $(k - num)$-th element, in other case we call recursively $mapped(i)$, $mapped(j)$, k on the left node. As for the base case we will just return the character which is in the leaf.

```
def quantile(self, k, l, r):
  if k < 1 or k > r-l+1:
    return None
  if self.leaf:
    return self.smallest if k <= self.n else None
  left_num = self.prefix_sum[r] - self.prefix_sum[l-1]
  if r-l+1-left_num >= k:
    new_l, new_r =  self._left_tree_range(l, r)
    return self.left.quantile(k, new_l, new_r)
  new_l, new_r =  self._right_tree_range(l, r)
  return self.right.quantile(k-r+l-1+left_num, new_l, new_r)
```

All the queries requires constant time in a single node and use one recursion call in one of the child, so the time complexity is proportional to the depth of the tree, which is $\mathcal{O}(\log(|\mathcal{A}|))$.

RangeCount query

As for `range_count(i, j, x, y)` query here we use another approach. For each range that fulfills condition $x \leq node.smallest$ and $node.biggest \leq y$ the answer is simple, it is $j - i + 1$. So we want to find the closest nodes that fulfill that condition and sum the answers. To find such nodes, we can use recursion in the following way (the order of conditions is important):

1. Base case: if $x \leq node.smallest$ and $node.biggest \leq y$, then return (j - i + 1)
2. If the node is a leaf, then return 0

3. If $[x, y]$ intersect with range of elements of left child and right child, then use recursive call on both children and return the sum of results.
4. If $[x, y]$ intersect only in the right child range of elements, then return the result of recursion call for right child.
5. Otherwise, return the result of recursion call for left child.

The intersection of ranges can be implemented naively, just considering all cases:

```
def _does_one_range_end_in_another(self, l, r, i, j):
  return (i <= l <= j) or (i <= r <= j)

def _ranges_intersect(self, l, r, i, j):
  return (self._does_one_range_end_in_another(l, r, i ,j) or
    self._does_one_range_end_in_another(i, j, l, r))
```

The first case is just a base case, so we return the length of the range. In the second case if the current node is leaf, then it means that no element fulfills the condition, so we terminate the recursion here. As for other case we just check if there are some elements in each child and invoke a proper recursion call. To prove the time complexity we need the following fact:

Observation 3. *After the first call of case* 3 *for the next calls of case* 3 *one of its child will fulfill base case condition.*

Proof. We inductively assume that this property holds for the children of the node. Each element of left child is smaller than any element of the right child. That implies that for each case 3 which occurs in the left child, the recursion call on the right child will fulfill the base case due to the fact that earlier in one of the ancestors of that node we used recursion call for right node, but all elements there were greater than in current child. Hence, that $node.right.biggest \leq y$ and also we used recursion call for the left side, but it can only happen if $x \leq node.left.biggest$, which implies that $x \leq node.right.smallest$, and it is exactly the base condition. The analysis for case 3 calls on the nodes on the right after the first occurrence of case 3 is similar. □

Using that observation we can now calculate properly the time complexity of `range_count(i, j, x, y)`. If case 3 did not occur at all during the query the answer is simple, it is just the height of tree which is $\mathcal{O}(\log(|\mathcal{A}|))$. In the other case the first case 3 occurs after at most $\mathcal{O}(\log(|\mathcal{A}|))$ operations, and now we can calculate the time needed for left and right child. On each of the next occurrences of case 3 we can observe that it will use $\mathcal{O}(1)$ operations for at least one of the child and in the other child in the worst case the recursion can go deeper which also can be represented as height of the tree which is $\mathcal{O}(\log(|\mathcal{A}|))$. In the worst case we need $\mathcal{O}(\log(|\mathcal{A}|))$ operations for the first occurrence and $\mathcal{O}(\log(|\mathcal{A}|))$ operations for both children which in total results in $\mathcal{O}(\log(|\mathcal{A}|))$ running time. The implementation of that function is shown below:

```python
def range_count(self, l, r, x, y):
  if l > r or l > self.n or l < 1 or x > y:
    return 0
  if x <= self.smallest  and self.largest <= y:
    return r-l+1
  if self.leaf or y < self.smallest or x > self.largest:
    return 0
  l_node, r_node = self.left, self.right
  if (self._ranges_intersect(l_node.smallest, l_node.largest, x, y) and
      self._ranges_intersect(r_node.smallest, r_node.largest, x, y)):
    new_left_l, new_left_r = self._left_tree_range(l, r)
    new_right_l, new_right_r = self._right_tree_range(l, r)
    return (self.left.range_count(new_left_l, new_left_r, x, y) +
      self.right.range_count(new_right_l, new_right_r, x, y))
  if (self._ranges_intersect(self.right.smallest,
      self.right.largest, x, y)):
    new_l, new_r = self._right_tree_range(l, r)
    return self.right.range_count(new_l, new_r, x, y)
  new_l, new_r = self._left_tree_range(l, r)
  return self.left.range_count(new_l, new_r, x, y)
```

Note that `rank(c, l, r)` and `range_count(l, r, x, y)` functions are easy to modify to return all indices. The only difference is to return range $[l, r]$ in the base case and map all indices from the recursion call to current node indices. Modification of `range_count(l, r, x, y)` returning all indices will be called `range_search(l, r, x, y)` and will be useful in construction of LZ-Index.

Chapter 2

FM-Index

2.1 Idea

The general idea of FM-Index [FM05] is quite simple. We will want to find a way to get a range of the suffix array such that the pattern occurs in all the suffixes in that range. It can be seen that if the pattern occurs in text t then it occurs in a consistent range on the suffix array of t. More formally:

Theorem 1. *If s occurs in t then there exists a range $[i, j]$ such that for all $k \in [i, j]$: $t[SA[k] : SA[k] + m] = s$ and for all $l \notin [i, j]$, it holds that $t[SA[l] : SA[l] + m] \neq s$.*

Proof. The proof follows from Observation 1. □

Earlier it was shown that such a range can be obtained using two binary searches over suffix array, but it requires in the worst case $\mathcal{O}(m \cdot \log(n))$ time. Using FM-index we can change that complexity. The required structure for FM-index is `RankSearcher` which can answer `prefix_rank(i, c)` query. `prefix_rank(i, c)` returns the number of characters c in prefix of length i for given text or array. Assuming that we can create `RankSearcher` in $\mathcal{O}(f(n, |\mathcal{A}|))$ time and time required for `prefix_rank(i, c)` is $\mathcal{O}(g(n, |\mathcal{A}|))$ time, the complexity of FM-index will be $\mathcal{O}(n + f(n, |\mathcal{A}|))$ time for creation and $\mathcal{O}(m \cdot g(n, |\mathcal{A}|))$ time for query. The possible structures that may be used to implement `RankSearcher` are as follows:

1. Wavelet Tree – it can be constructed in $\mathcal{O}(n \cdot \log(|\mathcal{A}|))$ time and can answer `rank(i, c)` query in $\mathcal{O}(\log(|\mathcal{A}|))$ time, which results in $\mathcal{O}(n \cdot \log(|\mathcal{A}|))$ time for creation of FM-index and $\mathcal{O}(m \cdot \log(|\mathcal{A}|))$ time to perform search. This structure is described in Section 1.4.3.

2. Prefix array for each character in $\mathcal{A}$ – this structure can be built in $\mathcal{O}(n \cdot |\mathcal{A}|)$ time but answers a query in $\mathcal{O}(1)$ time, that means we can create the FM-index in $\mathcal{O}(n \cdot |\mathcal{A}|)$ and perform a search in $\mathcal{O}(m)$ time. This structure is used more commonly in practice due to the small size of alphabet.

2.1.1 Idea of algorithm

Let us show the main idea on a following example: FM-index stores text $t = ababa\$$, suffix array of t, $BWT(t)$ and length of text n but as an example to better show the idea we will use the matrix from the BWT section. The pattern given for query is $s = aba$. So let us consider a matrix of circular shifts of t sorted in lexicographic order.

$$\begin{bmatrix} \$ababa \\ a\$abab \\ aba\$ab \\ ababa\$ \\ ba\$aba \\ baba\$a \end{bmatrix}$$

As it was show earlier the first column of matrix is the suffix array of t and the last column is $BWT(t)$. Now we want to search for the pattern. Let us consider a pattern character by character backwards, so the first character we want to use is **a** from $s = ab\mathbf{a}$. Now we want to find a range such that all the suffixes begin with that character (the details of how to do it fast will be explained later), in this case this range is [1,3].

$$\begin{bmatrix} \$ababa \\ \mathbf{a\$abab} \\ \mathbf{aba\$ab} \\ \mathbf{ababa\$} \\ ba\$aba \\ baba\$a \end{bmatrix}$$

The next operation to perform is to find the new range for the previous letter in pattern which is **b**, but it must be done in such a way that this range will only contain suffixes that starts with b and have the already performed suffix of pattern as a next character, so we want to find all corresponding suffixes which starts with **b** and are obtained from suffixes from previous range. So for all suffixes in range [1,3] they have to have **b** as the value of the last column (which means the previous character for that suffix was b). Only suffixes from the range [1,2] fulfill that condition.

$$\begin{bmatrix} \$ababa \\ \mathbf{a\$abab} \\ \mathbf{aba\$ab} \\ ababa\$ \\ ba\$aba \\ baba\$a \end{bmatrix}$$

Next, we have to map that range for the corresponding suffixes starting with **b**, which is the range $[4, 5]$. We will explain later a way of building such a range.

$$\begin{bmatrix} \$ababa \\ a\$abab \\ aba\$ab \\ ababa\$ \\ \mathbf{ba\$aba} \\ \mathbf{baba\$a} \end{bmatrix}$$

This way we construct the result range step by step. The algorithm ends after obtaining the range for the first character of the pattern which in this example is range $[2, 3]$, which is the correct answer.

$$\begin{bmatrix} \$ababa \\ a\$abab \\ \mathbf{aba\$ab} \\ \mathbf{ababa\$} \\ ba\$aba \\ baba\$a \end{bmatrix}$$

If an algorithm cannot create a proper range at any step the answer is that pattern does not exist in the text. The correctness of that process follows from the fact that we construct ranges step by step and in each step we obtain a proper range for each consecutive suffix of s.

2.1.2 API of FM-Index

The basic function of FM-Index is finding a range of suffix array such that the pattern occurs in all the suffixes in that range, but what can be obtained from that information.

- `count(s, m)` – Counting the number of occurrences of the pattern has $\mathcal{O}(m \cdot g(n, |\mathcal{A}|))$ time complexity. To obtain the number of occurrences from the given range of suffix array we just retrieve the size of that range and it will be the expected result.

- `exists(s, m)` – Checking if the pattern occurs in text in $\mathcal{O}(m \cdot g(n, |\mathcal{A}|))$ time. Getting the answer for that function is equivalent to checking if `count(s, m)` > 0.

- `all_occurrences(s, m)` – All occurrences of the pattern in text in $\mathcal{O}(m \cdot g(n, |\mathcal{A}|) + R)$ time complexity, where R is the number of occurrences of the pattern in text. To obtain such information you can just map indexes from given range to indexes at that positions in suffix array.

- `any_occurrence(s, m)` – Any occurrence of the pattern in text in $\mathcal{O}(m \cdot g(n, |\mathcal{A}|))$ time. It can be done by mapping any index from result range to index of suffix array.

- `first_occurence(s, m)` – Getting first occurrence of the pattern in text. There are many ways to get that, one of them require to get all occurrences and find the minimal index, but it requires $\mathcal{O}(m \cdot g(n, |\mathcal{A}|) + k)$ time, which at the worst case can be a lot. Finding it faster requires solving the RMQ (Range minimal query) problem, which can be achieved in many ways, using such data structures:

  - Sparse table [BF00] – which will require $\mathcal{O}(n \cdot \log n)$ preprocessing time and space, and it returns an answer to such query in $\mathcal{O}(1)$ time.
  - Segment tree [Zha19] – using segment tree allows us to compute the answer in $\mathcal{O}(\log n)$ time, but it will only require $\mathcal{O}(n)$ time and space for preprocessing.
  - Combined sparse table and segment tree – combining both of above techniques allows solving RMQ in $\mathcal{O}(\log \log n)$ for query and $\mathcal{O}(n \cdot \log \log n)$ time for preprocessing.
  - Optimal RMQ algorithm [BF00] (Sparse table + Cartesian trees + segmentation) – optimal algorithm finds an answer in $\mathcal{O}(1)$ time and requires only $\mathcal{O}(n)$ time for preprocessing.

- `last_occurrence(s, m)` – Getting the last occurrence of the pattern. It can be done similarly to `first_occurrence(s, m)` with the only change that the answer is maximum instead of minimum.

- `count_in_range(s, m, i, j)` – Getting the number of occurrences of s in substring $t[i:j]$. It can be done using separated Wavelet Tree on indices of SA using at most $\mathcal{O}(m \cdot g(n, |\mathcal{A}|) + \log(n))$ operations.

2.2 Details of preprocessing for FM-Index

The creation of FM-Index will require the text t, Burrows-Wheeler transform BWT of t, suffix array SA of t, *RankSearcher* structure, which can perform `prefix_rank(i, c)` query and *Beginnings* structure which will help to get a range in which character c occurs in a given array.

2.2.1 Creation of F and L

The array F will correspond to the first column of the matrix which was shown in Section 2.1.1, which is array of character created from the suffix array as $F[i+1] = t[SA[i]]$ and $F[0] = \$$. The array L corresponds to the last column of the previous matrix and will be equal to BWT. This stage takes at most $\mathcal{O}(n)$ time and space.

2.2.2 Creation of range mapping for character

The next structure, denoted as MAP, is a dictionary that maps characters from set $\{\$\} \cup \mathcal{A}$ to indices in the range $[0, |\mathcal{A}|]$. It can be done by iterating over L and assigning the first unused index for each distinct character.

The next step involves a construction of mapping from range $[0, |\mathcal{A}|+1]$ to the first occurrence of that character in F denoted as $FIRST$. It is known that all the same characters are in consistent range because, F was created based on suffix array. We built it by iterating over F and for each mapped character assign current index in F to our structure unless that character already has assigned index in mapping. Without loss of generality, we define $FIRST[|\mathcal{A}|] = n + 1$ to handle edge case where a character is the last one in mapping. On that index the character does not exist, but will mark the end of previous character occurrences. Therefore, the range in which character c exists in L is confined to $[FIRST[MAP[c]], [FIRST[MAP[c] + 1]] - 1]$

2.2.3 Creation of Beginnings structure

The next structure is very simple due to the dependence on the array F where all first letters are sorted. It is sufficient to iterate over the suffix array and assign the proper values.

```
def create_beginnings_structure(F, n):
    beginnings = [2]
    last = F[2]
    for i in range(3, n+2):
      if F[i] != last:
        last = F[i]
        beginnings.append(i)
    return beginnings
```

2.2.4 Creation of RangeSearcher

Here we will focus on the approach based on prefix array. In practice the alphabet is small, so this approach can be faster than using Wavelet Tree. To save some space we can also compute the prefix array only for some selected indices, so during the query we will have to expand our range like in naive the approach.

The code creating that data structure can be written as follows:

```
# t - text, n - lenght of t, A - alphabet of t
def create_range_searcher(t, n, A):
  result = dict()
  for character in A:
    prefix = []
    prefix.append(1 if t[0] == character else 0)
    for i in range(1, n+1):
      prefix.append(prefix[i-1] + (1 if t[i] == character else 0)
    result[character] = prefix
  return result
```

The query for that structure will look like on typical prefix array.

```
def rank(self, i, c):
    return self.prefix[c][i]
```

By combining all the algorithms above we obtain a complete FM-index structure.

2.3 Details of querying on FM-Index

The main function of FM-index is `_get_range_of_occurrences(s, k)` which will return range $[i, j]$ in suffix array where pattern s occurs. Like was described in the idea of that algorithm we will iterate over s backward and will store the proper range for current suffix of s. The only thing to explain is how to expand the current suffix of s. Let us suppose that we are currently at the index $i > 1$ in s and the current range is $[a, b]$. Then we:

1. Get the next character $c = s[i - 1]$.

2. Check if c exists in t, it can be done using character mapper.

3. Get number of characters c which are before l which is $x =$ `prefix_rank(l-1, c)` and number of characters c that are in prefix of length r which is $y =$ `prefix_rank(r, c)`, if $x = y$ that means that there is no character c in our range, so it can be stopped here.

4. Set $a =$ `beginnings[mapper[i]]` $+ x$, $b =$ `beginnings[mapper[i]]` $+ y - 1$ as the new range and decrement i.

Using that procedure we can now start properly for the first character and execute that procedure in loop to get an answer.

```
def _get_range_of_occurrences(FM, p, size):
  if size > FM.n or size == 0:
    return -1, -1
  if p[-1] not in FM.mapper_of_chars:
    return -1, -1
  map_idx = FM.mapper_of_chars[p[-1]]
  l= FM.beginnings[map_idx]
  r = (FM.beginnings[map_idx + 1] - 1
    if map_idx != FM.len_of_alphabet - 1 else FM.n + 1)
  for c in p[-2:0:-1]:
    if c not in FM.mapper_of_chars:
      return -1, -1
    occurrences_before = FM.rank_searcher.prefix_rank(c, l - 1)
    occurrences_after = FM.rank_searcher.prefix_rank(c, r)
    if occurrences_before == occurrences_after:
      return -1, -1
    map_idx = FM.mapper_of_chars[c]
    l = FM.beginnings[map_idx] + occurrences_before
    r = FM.beginnings[map_idx] + occurrences_after - 1
    if r < l:
      return -1, -1
  return l, r
```

Chapter 3

LZ-Index

3.1 Preliminaries

3.1.1 LZ compression

The most important part in LZ-Index [Nav04] is the fact that we want to search for patterns within the compressed text. The compressions that can be used for this particular structure are LZ78 [LZ78] and LZW [Ter84]. Here we will focus on the LZ78 compression.

Idea of LZ78 compression

The main idea of LZ78 compression is to convert text t into sequence of blocks, where the block will be in form (i, c) where i is an index and c is the character of t, so the output of the LZ78 compression will be a sequence of $B_1, ..., B_w$. Each of the block will clearly represent some substring of t and after translating this block to that substring we will get exactly text t. The block B_x is defined as a pair (i, c) such that B_x correspond to the string obtained from decompressing B_i adding character c at the end or just c if $i = 0$. So we will use the previous blocks to create a new one by extending them with one character, so the idea can be written as:

1. Start with the empty string q and previous block as $prev = 0$.
2. Get next character c of t.
3. If there is no block that represents string $q + c$ or c is last character of t, then add to result block $(prev, c)$, set $prev = 0$ and go to step 3.
4. Otherwise, which means that there is a block B_z, that it represents string $q + c$, set $prev = z$ and go to step 3.

To summarize it with one sentence: get next character of t and add it to current string, if it is unique at that moment create a new block from the previous one and start again from empty string with next character of t.

Example of LZ78 compression

Let's try to find a LZ78 compression of text $t = aabb$. We start with:

step	result	current string	previous block	blocks mapping
0	[ ]	' '	0	{}

Going forward with loop the next character is a.

step	result	current string	previous block	blocks mapping
0	[ ]	' '	0	{}
1	[ ]	'a'	0	{}

There is no block that correspond to the string currently under consideration so we add new block, set default values and look at the next character.

step	result	current string	previous block	blocks mapping
0	[ ]	' '	0	{}
1	[ ]	'a'	0	{}
2	[(0, a)]	'a'	0	{ B_1 = 'a' }

There is already a block that correspond to string $'a'$, so we continue with the next character.

Step	result	current string	previous block	blocks mapping
0	[ ]	' '	0	{}
1	[ ]	'a'	0	{}
2	[(0, a)]	'a'	0	{ B_1 = 'a' }
3	[(0, a)]	'ab'	1	{ B_1 = 'a' }

Now string currently under consideration is unique withing current mapped blocks, so we add it as a new block and goes with new character.

step	result	current string	previous block	blocks mapping
0	[ ]	' '	0	{}
1	[ ]	'a'	0	{}
2	[(0, a)]	'a'	0	{ B_1 = 'a' }
3	[(0, a)]	'ab'	1	{ B_1 = 'a' }
4	[(0, a), (1, b)]	'b'	0	{ B_1 = 'a', B_2 = 'ab' }

The last character is b which is also unique string, so the result of compression will be $(0, a), (1, b), (0, b)$.

Properties of LZ78 compression

Let us start by stating some interesting properties of the LZ78 compression.

Observation 4. *[Nav04] Let $B_1, ..., B_w$ be the LZ78 compression of t. Each of these blocks is unique, possibly except the last one. We can make them all unique appending to t a character that does not appear in the alphabet.*

Proof. Let us prove it by contradiction. Let us assume that there are two blocks B_i, B_j such that $B_i = B_j$, $1 \leq i \neq j < w$. Without loss of generality we assume that $j > i$. Taking into consideration LZ78 compression algorithm. If there already exists the block that is representing the same substring, we are considering the next character of t, so we cannot add that block to the result. From the fact that $i, j < w$, none of the blocks B_i, B_j is the last one block, so the next character always exists. That implies that such situation cannot happen. Appending the character that does not appear in the alphabet, we ensure that the last one block is unique, because such character is already unique in t. □

The above observation is essential in LZ-index pattern searching procedure, we will need all the blocks to be unique.

Observation 5. *[LZ78] LZ78 is a lossless compression, that is, it can be decompressed unambiguously, to obtain text t.*

That observation can be easily by contradiction straight form the decompressing procedure.

Observation 6. *[Goe15] In the worst case LZ78 compression will use $x + o(x)$ bits, where x is a number of bits of uncompressed text.*

Observation 7. *[Goe15] The compression rate of LZ78 compression in average case asymptotically approaches to entropy of text t.*

Proof of above observation is quite complicated and requires advanced knowledge of probability, so it won't be discussed here.

3.1.2 LZTrie

The first structure required for LZ-Index is LZTrie. It is a tree consisting of all blocks of LZ78 compression, where each node correspond to a single block of compressed text. By creating that structure you can also compress the text at once, like in the compression procedure presented above. The only difference in this case is that all blocks are represented as nodes. That structure has to permit the following operations for each node:

- `id(x)` – returns the identifier of block which can be just an index of the block in compressed text.

- `children(x)` – computes a structure that holds all the children of given node. The returned collection has to answer fast for query like `child(c)` which should return a child of x such that string represented by x is extended by character c in that child or recognize that such child not exist.

- `parent(x)` – provides the parent of given node.

- `depth(x)` – retrieves the depth of given node counted from the root of the tree. It will be used to count the offset for pattern beginning index.

- `rank(x)` – returns the rank of given node in lexicographic order.

- `position(x)` – returns a position in which given node begins in uncompressed text. It is used for mapping the raw result of string-searching performed on LZ-Index, which is a set of pairs (i, j), where i is the index of a block in compressed text and j is the offset for the beginning of that block.

- `left_rank(x)` and `right_rank(x)` – retrieves the minimum and maximum rank of proper node in given node subtree.

The LZTrie structure has to perform search for given string that will return the node which represents that string in LZTrie.

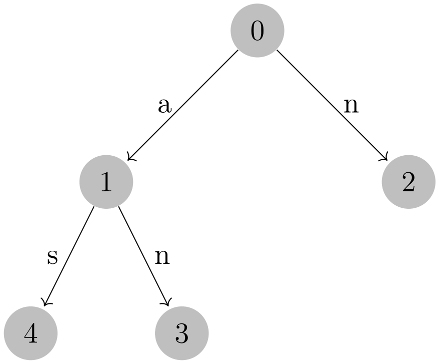


Figure 3.1: Example LZTrie for $t = ananas$

The creation of LZTrie node and also RevLZTrieNode can be implemented in the following way:

```
class _LZTreeNode:
  def __init__(self, parent, character, idx, position):
    self.parent = parent
    self.position = position
    if parent is not None:
      parent.children[character] = self
      self.depth = parent.depth + 1
    else:
      self.depth = 0
    self.idx = idx
    self.children = {}
    self.character = character
    self.rank = None
    self.left_rank = None
    self.right_rank = None
```

For proper setting $rank$, $left\_rank$, $right\_rank$ properties we have to build the whole tree and then traverse it, so it requires separate function:

```
  def set_ranks(self, rank):
    if self.idx is not None:
      self.rank = rank
      self.left_rank = rank
      self.right_rank = rank
      rank = rank + 1
    if len(self.children) > 0:
      for child_key in sorted(self.children):
        rank = self.children[child_key].set_ranks(rank)
      min_key = min(self.children)
      max_key = max(self.children)
      self.left_rank = (self.children[min_key].left_rank
        if (self.rank is None or
          self.children[min_key].left_rank < self.rank)
        else self.rank)
      self.right_rank = (self.children[max_key].right_rank
        if (self.rank is None or
          self.children[max_key].right_rank > self.rank)
        else self.rank)
    return rank
```

As for implementation of construction the LZTrie it looks just like the compression algorithm mentioned in Section 3.1.1.

```
class _LZTrie:
  def __init__(self, t, n):
    t += '$'
    self.root = _LZTreeNode(None, '#', 0, None)
    current_node = self.root
    index, position = 1, 1
    for i in range(1, n+2):
      current_char = t[i]
      if current_char not in current_node.children:
        _LZTreeNode(current_node, current_char, idx, position)
        index += 1
        current_node = self.root
        position = i+1
      else:
        current_node = current_node.children[current_char]
    self.size = index
    self.root.set_ranks(0)

def search(tree, t, n):
  return _search_private(t, 0, n, tree.root)

def _search_private(tree, index, n, node):
  if index == n:
    return node
  if tree[index+1] not in node.children:
    return None
  return _search_private(tree, index+1, n, node.children[tree[index+1]])
```

LZTrie creation requires $\mathcal{O}(n \cdot \log(|\mathcal{A}|))$ time for a given text of length n.

3.1.3 RevLZTrie

RevLZTrie structure behaves like LZTrie, but there are some differences. The first one is that the RevLZTrie is created by passing each node of LZTrie and going from it to the root. Formally, it is created by reversed strings of compressed text and not by passing the reversed text to LZTrie. The RevLZTrie consists of the same types of nodes as LZTrie so each of them can perform the same operations. The main difference is that in RevLZTrie there are some nodes that do not correspond to any blocks of the compressed text, so the size of that structure can be equal to the length of uncompressed text, but still the complexity of creation it is exactly the same as for LZTrie which is $\mathcal{O}(n \cdot \log(|\mathcal{A}|))$.

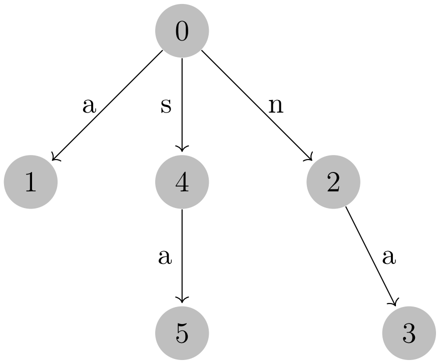


Figure 3.2: RevLZTrie for LZTrie from Figure 3.1

The example implementation looks like this:

```
class _RevLZTrie:
  def __init__(self, lz_trie):
    self.root = _LZTreeNode(None, '#', 0, None)
    self._add_recursive(lz_trie.root)
    self.root.set_ranks(0)

  def _add_recursive(self, node):
    for child in node.children.values():
      self._add_recursive(child)
      self._add_block(child, self.root, child.idx)

  def _add_block(self, lz_node, rev_node, idx):
    if lz_node.parent is None or lz_node.parent.character == '#':
      if lz_node.character in rev_node.children:
        rev_node.children[lz_node.character].idx = idx
      else:
        rev_node.children[lz_node.character] = (_LZTreeNode(rev_node,
          lz_node.character, idx, None))
    else:
      if lz_node.character not in rev_node.children:
        rev_node.children[lz_node.character] = (_LZTreeNode(rev_node,
          lz_node.character, None, None))
      self._add_block(lz_node.parent,
        rev_node.children[lz_node.character], idx)
```

Note that `search(tree, n, root)` function works properly for both LZTrie and RevLZTrie.

3.1.4 Range structure

One of the structures needed for LZ-index is a structure that can perform two-dimensional searching in space $[0, n] \times [0, n]$.

Problem 5 (Two-dimensional search in space $[0, n] \times [0, n]$). *Given a set of points* $P = \{(x, y) : x \in [0, n], y \in [0, n]\}$ *of size* m*. For each query of form* $[l_1, r_1]$*,* $[l_2, r_2]$ *return all points from* P *that* $x \in [l_1, r_1]$ *and* $y \in [l_2, r_2]$*.*

There are many ways to solve that problem, the first one is naive approach which for each point $p \in P$ checks if it fulfills condition from the query. Such approach is very simple and will work in $\mathcal{O}(m)$ time as for each point we need constant time to check the query condition, but it can be done faster. First, we sort the points by their first coordinate. Then, we create Wavelet Tree from the list of second coordinates from sorted points. With such preprocessing, we can now answer queries by performing two binary searches to first find the indices l, r such that all points in range $[l, r]$ fulfills search condition for first coordinate, and then we use `range_search(l, r, x, y)` function for Wavelet Tree with arguments l,

r, l_2, r_2 and maps the result. Note that the result of `range_search(l, r, x, y)` is exactly the list of all indices for points that fulfil the condition for first and second coordinate. Implementation of that structure can be done in the following way:

```python
class _RangeSearcher:
  def __init__(self, points):
    self.points = sorted(points, key= lambda x: x[0])
    values = ['#'] + [y for x, y in self.points]
    self.wavelet_tree = wavelet_tree.WaveletTree(values, len(values)-1)

  def search_in_range(self, l1, r1, l2, r2):
    l, r = 0, len(self.points)
    while l < r:
      s = (l+r)//2
      x, _ = self.points[s]
      if x < l1:
        l = s+1
      else:
        r = s
    left = l
    l, r = -1, len(self.points)-1
    while l < r:
      s = (l+r+1)//2
      x, _ = self.points[s]
      if x <= r1:
        l = s
      else:
        r = s-1
    right = l
    if left > right or left == len(self.points) or right == -1:
      return []
    return ([self.points[x-1] for x in
              self.wavelet_tree.range_search(left+1, right+1, l2, r2)])
```

Time needed for construction RankSearcher is equal to $\mathcal{O}(m \log m)$ due to the sorting and the creation of Wavelet Tree. Moreover, answering the query use only $\mathcal{O}(R \cdot \log(m) + \log(m))$ time, where R is the number of points that are returned from query. The input for creation of RangeSearcher will be all pairs in form $(B_i.rev\_rank, B_{i+1}.rank)$ for all i less than number of block in compressed text and where $B_i.rev\_rank$ is the rank if i-th block in RevLZTrie.

3.1.5 NodeMapper structure

The NodeMapper structure retrieves node of RevLZTrie that represents a block at the given index in LZ78 compression. Such structure can be created by using information from RevLZTrie nodes and just all of them in any order. It requires $\mathcal{O}(n)$ time due to the fact that some nodes of RevLZTrie does not represent any block of compressed text.

```
class _NodeMapper:
  def __init__(self, lz_trie, size):
    self.arr = [None] * size
    self._map_tree_to_list(lz_trie.root)

  def _map_tree_to_list(self, node):
    if node.idx is not None:
      self.arr[node.idx] = node
    for child in node.children.values():
      self._map_tree_to_list(child)

  def get_node_by_idx(self, idx):
    return self.arr[idx]
```

The creation of this structure requires to visit all $\mathcal{O}(n)$ nodes of RevLZTrie, and the query requires $\mathcal{O}(1)$ time. This structure is also used for receiving the text position from blocks, so it also has to be build for LZTrie.

3.1.6 RankMapper structure

The last structure we need to perform string-searching in LZ78 compressed text is called RankMapper. That structure allows us to find the i-th node in lexicographical order. That structure has to be build for both LZtrie and RevLZTrie, but it can be also done in simple way using the idea already invoked for the creation of NodeMapper. To build RankMapper one can just visit all the nodes of the tree and set their index in a suitable array, as presented below:

```
class _RankMapper:
  def __init__(self, lz_trie, size):
    self.arr = [None] * size
    self._map_tree_to_list(lz_trie.root)

  def _map_tree_to_list(self, node):
    if node.rank is not None:
      self.arr[node.rank] = node
    for child in node.children.values():
      self._map_tree_to_list(child)

  def get_node_by_rank(self, rank):
    return self.arr[rank]
```

The complexity of creation RankMapper and performing a query is exactly the same as for NodeMapper. Having all of that structures in hand, LZ-Index can be constructed as follows:

```
class _LZIndex:
  def __init__(self, lz_trie, rev_lz_trie, lz_node_mapper,
        rev_lz_node_mapper, range_searcher, lz_rank_mapper,
        rev_lz_rank_mapper):
    self.lz_trie = lz_trie
    self.rev_lz_trie = rev_lz_trie
    self.lz_node_mapper = lz_node_mapper
    self.range_searcher = range_searcher
    self.rev_lz_node_mapper = rev_lz_node_mapper
    self.lz_rank_mapper = lz_rank_mapper
    self.rev_lz_rank_mapper = rev_lz_rank_mapper

def create_lz_index(t, n):
  lz_trie = _LZTrie(t, n)
  rev_trie = _RevLZTrie(lz_trie)
  lz_node_mapper = _NodeMapper(lz_trie, lz_trie.size)
  rev_node_mapper = _NodeMapper(rev_trie, lz_trie.size)

  points = [(rev_node_mapper.get_node_by_idx(i).rank,
    lz_node_mapper.get_node_by_idx(i+1).rank)
    for i in range(1, lz_trie.size - 1)]
  range_searcher = _RangeSearcher(points)
  lz_rank_mapper = _RankMapper(lz_trie, lz_trie.size)
  rev_lz_rank_mapper = _RankMapper(rev_trie, lz_trie.size)
  return _LZIndex(lz_trie, rev_trie, lz_node_mapper, rev_node_mapper,
    range_searcher, lz_rank_mapper, rev_lz_rank_mapper)
```

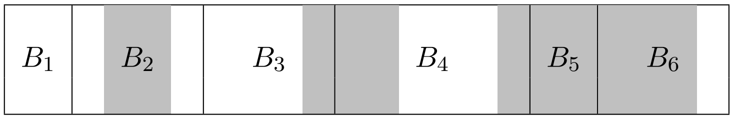
3.2 Idea of LZ-Index searching

String-searching in compressed text using LZ78 compression may be split in three cases depending on the number of block in which the pattern is located.

1. Pattern occurrence is fully located inside a single block.
2. Pattern occurrence lies in to following blocks, it means that the prefix of the pattern matches the suffix of some block B_i and the left suffix of the pattern matches with the prefix of the block B_{i+1} for some i.

3. Pattern occurrence lies in three on more blocks, which means that it will be like in case 2, but there can be some blocks in between.

Types of occurrences



Let us consider all these cases in turn.

3.2.1 Occurrences inside one block

To find all occurrences in that case we look at blocks as the independent texts. We use the idea of searching in the suffix array, that is, we want to obtain all suffixes for each block and check if the pattern is its prefix. It can be done all at once, so we do not have to get all suffixes for each block. To achieve that we use RevLZTrie, but first we have to notice some properties. We know that if there is some block B_i in LZTrie, then LZTrie also contains blocks that corresponds to all prefixes of the substring compressed in B_i. That property will follow to the observation below.

Observation 8. *Let B_j be the parent block of B_i. If a pattern s occurs inside the string compressed in B_i, and it is not a suffix of string compressed in B_i then string compressed in B_j also contains the pattern s.*

That means if we find the node corresponding to s in RevLZTrie, denoted as v, then all nodes that correspond to some block in the subtree of v will be a candidate to be the pattern occurrence, but it will be even a little more form that observation. For each correct node u in subtree of v it is enough to find it in LZTrie and all nodes in subtree of u will correspond to an occurrence. This follows to the algorithm for the first case.

1. Find the node v that correspond to s^r in `RevLZTrie`.
2. For each i in range $[x.left\_rank, x.right\_rank]$ do:
   (a) Find node u in LZTrie that correspond to the node with rank i in RevLZTrie, it can be done by using RevLZRankMapper and LZNodeMapper.
   (b) For each node x in subtree of u add to result index $x.position + u.depth - m$.

Finding the node v requires at most $\mathcal{O}(m)$. From Observation 8. follows that for each node in subtree of v there is at least one unique occurrence. That gives us that by traversing the subtree of v we obtain all occurrences of that type. This gives us in total $\mathcal{O}(m + R)$ time complexity. In short, the above procedure can be implemented with all edge cases handled as follows:

```
def _contains_in_single_block(lz_index : _LZIndex, s, m):
  v = '#' + (s[::-1])[:-1]
  root = search(lz_index.rev_lz_trie, v, m)
  if root is not None:
    for i in range(root.left_rank, root.right_rank + 1):
      rev_node = lz_index.rev_lz_rank_mapper.get_node_by_rank(i)
      node = lz_index.lz_node_mapper.get_node_by_idx(rev_node.idx)
      for j in range(node.left_rank, node.right_rank + 1):
        result_node = lz_index.lz_rank_mapper.get_node_by_rank(j)
        yield result_node.position + node.depth - m
```

3.2.2 Occurrences spanning two blocks

In the second type of occurrences we consider all partitions of the pattern. It was mentioned before that the occurrences are in form such that the prefix of the pattern is the suffix of one block and the suffix of the pattern is the prefix of the next block. Therefore, for each partition of the pattern we find all nodes that contains the first part of partition as a suffix, which means searching for the reversed first part in RevLZTrie and for the second part in LZTrie. Let us denote u as the result of search for the prefix of the pattern and v the result of search for the suffix of the pattern. Now we want to check that if there are nodes x, y such that x is in subtree of u, y is in subtree of v and $x.idx + 1 = y.idx$. That way the nodes x, y are neighbors. We can find such nodes using RangeSearcher structure in the following way.

1. For all i in range $[1, |s| - 1]$ do:
   (a) Let $pref$ be the prefix of s of length i and suf the suffix of s of length $|s| - i$.
   (b) Find node that correspond to reversed $pref$ in RevLZTrie, let us denote the result as u and find node that correspond to the suf in LZTrie, let us denote it as v.
   (c) For all points (x, y) returned by RangeSearcher query with arguments $u.left\_rank$, $u.rigth\_rank$, $u.left\_rank$, $v.right\_rank$. Let us denote z as the node of LZTrie that corresponds to the node with rank x in RevLZTrie and add to result index $z.position + z.depth - i$.

For each position index in patter we perform a query on RangeSearcher and return the results, so it is easy to observe that the time complexity is $\mathcal{O}(m^2 + m \cdot \log(w) + R \cdot \log(w))$. Above procedure can be implemented in the following way:

```
def _contains_within_two_blocks(lz_index : _LZIndex, s, m):
  for i in range(1, m):
    rev_prefix = '#' + (s[::-1])[m-i:m]
    sufix = '#' + s[i+1:]
    rev_node = search(lz_index.rev_lz_trie, rev_prefix, i)
    node = search(lz_index.lz_trie, sufix, m-i)
    if rev_node is None or node is None:
      continue
    l1,r1 = rev_node.left_rank, rev_node.right_rank
    l2,r2 = node.left_rank, node.right_rank
    for (x, _) in (lz_index.range_searcher.search_in_range(l1,
            r1, l2, r1)):
      rev_node = lz_index.rev_lz_rank_mapper.get_node_by_rank(x)
      node = lz_index.lz_node_mapper.get_node_by_idx(rev_node.idx)
      yield node.position + node.depth - i

```

3.2.3 Occurrences spanning three or more blocks

For the last case we will use the property of LZ78 compression that all the block in result compression are unique to make an observation.

Observation 9. *[Nav04] There is only one block B_i such that the string compressed in B_i is equal to $s[j:k]$.*

Proof. We prove it by contradiction: if there were two blocks that match such substring it would mean that they are the same, which conflicts with a property of LZ78 compression that all blocks corresponds to unique substrings. □

From that observation it can be also concluded that there are at most $\mathcal{O}(m^2)$ blocks that match some substring of s in that case. It follows from the fact that there are at most $\mathcal{O}(m^2)$ substring in s.

Observation 10. *If the substring compressed in block B_k lies inside the occurrence of the pattern, and it matches substring $s[i:j]$. It will not be used to match that substring in any other occurrence.*

Proof. Again we prove it by contradiction. Suppose that there are two different occurrences of s that contains block B_k at exactly position i in the pattern, which means that the next block are implied and the previous ones also because the next one has to be b_{k+1}, and it is also at the same position which means that these occurrences will be the same. □

Using these observations we compute array $C[i][j]$, which will store a node that correspond to the substring $s[i:j+1]$, array of dictionaries $A[i]$, which store a dictionary that maps the node index to an index j such that $C[i][j]=x$, array $visited[i][j]$, which store information that the block, which strings match $s[i:j+1]$ was already used. To compute these structures we use the following procedure:

1. For each i in range $[1,|s|]$ do:

   (a) Denote $LZTrie.root$ as v, and create empty dictionary d.

(b) For each j in range $[i, |s|]$ do:

   i. If $s[j]$ not in $v.children$ then go to step 1

   ii. Set $v = v.children[s[j]]$, $C[i][j] = v$ and $d[v.idx] = j$

(c) Set $A[i] = d$

Above procedure requires $\mathcal{O}(m^2)$ time and space and can be implemented as follows:

```python
def _prepare_structures_for_third_case(lz_index : _LZIndex, s, m):
  used = [[False]*(m+1) for _ in range(m+1)]
  existance = [[None]*(m+1) for _ in range(m+1)]
  arr = [{}]
  for i in range(1, m+1):
    recorded = {}
    current_node = lz_index.lz_trie.root
    for j in range(i, m+1):
      if (current_node is not None and
            s[j] not in current_node.children):
        current_node = None
      elif current_node is not None:
        current_node = current_node.children[s[j]]
      existance[i][j] = current_node
      if current_node is not None:
        recorded[current_node.idx] = j
    arr.append(recorded)
  return used, existance, arr
```

Using that information we can now search for all occurrences with case 3.

1. For all i in range $[1, |s|]$ do:

   (a) For all j in range $[i, |s|]$ do:

      i. If $C[i][j]$ is $None$ or $visited[i][j]$ is true, go back to step 2

      ii. Try to expand range $[i, j]$ with the next blocks from $C[i][j]$ that matches exactly as far as it's possible and denote the new range as $[i, r]$

      iii. If the next block does not contain $s_{r+1}, ..., s_m$ as a suffix, then go back to step 2

      iv. If the suffix of block with $idx = C[i][j].idx - 1$ contains $s_1, ..., s_{i-1}$ as a suffix and there are at least 3 blocks, then denote v as $LZNodeMapper.get\_node\_by\_idx(C[i][j].idx - 1)$ and add index $v.position - i + 1$ to the result.

During that process we mark every used node and check if it was not reused. This procedure can be done in $\mathcal{O}(m^3)$ time. Unfortunately, there are some edge cases which has to be considered within that procedure. Implementation that cover all of that edge cases can be seen below:

```
def _contains_within_three_or_more_blocks(lz_index : _LZIndex, s, m):
  used, exist, arr = _prepare_structures_for_third_case(lz_index, s, m)
  for i in range(1, m+1):
    for j in range(i, m+1):
      if exist[i][j] is None or used[i][j] is True:
        continue
      start_idx = exist[i][j].idx
      current_idx = start_idx
      current_end = j
      while current_end < m and (current_idx + 1) in arr[current_end+1]:
        current_idx = current_idx + 1
        used[current_end + 1][arr[current_end + 1][current_idx]] = True
        current_end = arr[current_end + 1][current_idx]
      size = current_idx - start_idx  + 1
      if i > 1:
        size = size + 1
      if current_end < m:
        size = size + 1
      if size < 3 or (current_end != m and
            exist[current_end+1][m] is None):
        continue
      if (lz_index.lz_trie.size > current_idx + 1 and
          (current_end == m or (exist[current_end+1][m].left_rank <=
          lz_index.lz_node_mapper.get_node_by_idx(current_idx+1).rank <=
          exist[current_end+1][m].right_rank ))):
        if i == 1:
          yield lz_index.lz_node_mapper.get_node_by_idx(start_idx).position
          continue
        if start_idx == 1:
          continue
        current_node = lz_index.lz_node_mapper.get_node_by_idx(start_idx-1)
        prev = i - 1
        while (prev > 0 and current_node.parent is not None and
            s[prev] in current_node.parent.children and
            current_node.parent.children[s[prev]] == current_node):
          prev = prev - 1
          current_node = current_node.parent
        if prev == 0:
          node = lz_index.lz_node_mapper.get_node_by_idx(start_idx)
          yield node.position - i + 1
```

By summing the complexities of all cases we get the total running time $\mathcal{O}(m^3 + (m + R) \cdot \log(w))$, where R is the number of occurrences of s in t. Note that some operations on

LZTrie and RevLZTrie use dictionaries, e.g. for `children(x)`, which in the worst case may require more than $\mathcal{O}(1)$ operations. If we want to use data structures with deterministic guarantees, such as balanced BST, the total complexity will be $\mathcal{O}(m^3 \cdot \log(|\mathcal{A}|) + (m + R) \cdot \log(w))$.

Chapter 4

Implementation and comparison of LZ-Index and FM-Index algorithms

4.1 Implementation

The algorithms discussed in this work were implemented in python. The source code of the implementation is publicly available in the **string-algorithms** GitHub repository and can be accessed at `https://github.com/krzysztof-turowski/string-algorithms/pull/57`.

4.1.1 Implementation details

Implementation of discussed structures is split into files. We outline the primary files, their content and usages below:

- `common/wavelet_tree.py` – File containing the implementation of Wavelet Tree structure.
- `string_indexing/fm_index.py` – File containing the FM-Index implementation, all required classes and creation methods.
- `string_indexing/lz_index.py` – File containing the implementation of the LZ-Index, all required structures and creation methods.
- `test/test_wavelet_tree.py` – Unit tests that ensure the correctness of the wavelet tree.
- `test/test_exact_string_matching.py` – Unit tests that ensure the correctness of implemented string-matching algorithms.

4.2 Comparison of FM-Index and LZ-Index

We will compare FM-Index and LZ-Index in terms of peak memory used, time needed and accesses to their structures. As for test cases, we will use test cases with different sizes of alphabet, size of inputs and with non-uniform characters distributions. Details description of all test cases are as follows:

- Small alphabet – In that test case we will focus on properties that described structures hold for small alphabet. In practice, we set the size of alphabet to 4 and generate strings over that alphabet.
- Medium alphabet – In that test case, we generate the words over the alphabet of size 16. We also choose characters of text and patterns with uniform distribution over $\mathcal{A}$. That test case will also provide the differences in performance in comparison to the small alphabet.
- Large alphabet – That test case have the size of alphabet fixed to 64. The text and patterns were still chosen randomly with uniform distribution over $\mathcal{A}$. It also provides data of comparison to smaller sizes of alphabets.
- Alphabet with non-uniform distribution with parameter 1/2 – That test case will stand out from the previous in terms of the probability, that each character is chosen from the alphabet to text and patterns. The alphabet used here will be infinite and the probability of choosing the i-th character form the alphabet is $(\frac{1}{2})^i$.
- Alphabet with non-uniform distribution with parameter 1/3 – This test case is similar to the previous test case. The only difference is that all characters from the alphabet are chosen with probability $(\frac{1}{3})^i$, for $i > 1$, and $\frac{5}{6}$, for $i = 1$, to ensure that the sum of probabilities is equal to 1.

4.2.1 Tests for small alphabet

Peak memory comparison

As for the peak memory for the creation of structures and the search, the results are as follows:

Table 4.1: Memory peak during structure creation for small alphabet

n	LZ-Index	Standard FM-Index	FM-Index with Wavelet Tree
10^2	103 KB	116 KB	38 KB
10^3	765 KB	921 KB	392 KB
10^4	6 MB	8 MB	4 MB
10^5	58 MB	77 MB	41 MB
10^6	534 MB	708 MB	391 MB
$3 \cdot 10^6$	2018 MB	2103 MB	1154 MB

Summarizing the result of this case for memory performance, all of these structures require nearly the same memory for creation. The exception is FM-Index with Wavelet Tree implementation, which uses around 2 times less memory.

Table 4.2: Memory peak during search for small alphabet

n	m	LZ-Index	Standard FM-Index	FM-Index with Wavelet Tree
10^2	10^1	6 KB	1443 B	1056 B
10^3	10^1	6 KB	1530 B	1059 B
10^3	10^2	194 KB	1623 B	1152 B
10^4	10^1	6 KB	1601 B	1220 B
10^4	10^2	195 KB	1688 B	1300 B
10^5	10^2	203 KB	1693 B	1311 B
10^6	10^2	204 KB	1694 B	1309 B
10^6	10^3	15 MB	2 KB	2 KB
10^6	10^4	1533 MB	11 KB	10 KB
$3 \cdot 10^6$	10^4	1533 MB	11 KB	10 KB

Memory performance of searching in this case shows that both implementations of FM-Index needs nearly the same amount of memory. However, the LZ-Index search always requires $\mathcal{O}(m^2)$ space, which results in much more memory used in practice.

Time comparison

The time needed for creation of structures and the search presents as follows:

Table 4.3: Time needed to create structures for small alphabet

n	LZ-Index	Standard FM-Index	FM-Index with Wavelet Tree
10^2	2 ms	1 ms	1 ms
10^3	21 ms	8 ms	8 ms
10^4	187 ms	96 ms	131 ms
10^5	1859 ms	1027 ms	1319 ms
10^6	19.45 s	9.8 s	15.13 s
$3 \cdot 10^6$	76.83 s	30.47 s	44.68 s

Time needed for construction standard FM-Index is the lowest and is 2 times faster than construction the LZ-Index and almost 1.5 faster than construction of the FM-Index with Wavelet Tree implementation.

Table 4.4: Time needed to perform search for small alphabet

n	m	LZ-Index	Standard FM-Index	FM-Index with Wavelet Tree
10^2	10^1	1 ms	1 ms	1 ms
10^3	10^1	1 ms	1 ms	1 ms
10^3	10^2	2 ms	1 ms	1 ms
10^4	10^1	2 ms	1 ms	1 ms
10^4	10^2	3 ms	1 ms	1 ms
10^5	10^2	3 ms	1 ms	1 ms
10^6	10^2	3 ms	1 ms	1 ms
10^6	10^3	759 ms	2 ms	2 ms
10^6	10^4	76.61 s	3 ms	2 ms
$3 \cdot 10^6$	10^4	79.03 s	3 ms	3 ms

Time performance of both implementation of the FM-Index is the same. Unfortunately, time needed by the LZ-Index to find all occurrences is huge due to at least $\mathcal{O}(m^2)$ time complexity in the best case.

4.2.2 Alphabet of medium size

Peek memory comparison

Table 4.5: Memory peak during structure creation for medium alphabet

n	LZ-Index	Standard FM-Index	FM-Index with Wavelet Tree
10^2	666 KB	939 KB	676 KB
10^3	2 MB	5 MB	7 MB
10^4	7 MB	41 MB	70 MB
10^5	26 MB	337 MB	685.0 MB
10^6	129 MB	3.12 GB	6.52 GB
$3 \cdot 10^6$	302 MB	9.01 GB	16.89 GB

The result of memory performance for medium alphabet are very different from for small alphabet. In that case memory used to construct the LZ-Index is around 30 times smaller than for the standard FM-Index and around 60 times smaller than for the FM-Index with Wavelet Tree implementation.

Table 4.6: Memory peak during search for medium alphabet

n	m	LZ-Index	Standard FM-Index	FM-Index with Wavelet Tree
10^2	10^1	524 KB	2 KB	2 KB
10^3	10^1	757 KB	17 KB	17 KB
10^3	10^2	44 MB	15 KB	15 KB
10^4	10^1	1289 KB	162 KB	162 KB
10^4	10^2	52 MB	161 KB	161 KB
10^5	10^2	85 MB	1563 KB	1563 KB
10^6	10^2	89 MB	1590 KB	1591 KB
10^6	10^3	1.2 GB	2 MB	2 MB
10^6	10^4	13.4 GB	4 MB	4 MB
$3 \cdot 10^6$	10^4	15 GB	4.2 MB	4.1 MB

Memory performance of obtaining all occurrences for both implementation of the FM-Index are similar, but for the LZ-Index it is still huge amount of memory needed.

Time comparison

Table 4.7: Time needed to create structures for medium alphabet

n	LZ-Index	Standard FM-Index	FM-Index with Wavelet Tree
10^2	18 ms	34 ms	25 ms
10^3	81 ms	367 ms	335 ms
10^4	409 ms	3.92 s	3.6 s
10^5	2.79 s	39.91 s	38.11 s
10^6	22.3 s	390 s	378 s
$3 \cdot 10^6$	56.3 s	18.9 min	16.76 min

In this case, time needed for construction of the LZ-Index is around 17 time less than both implementations of the FM-Index. It is very different in comparison with the construction for the small alphabet.

Table 4.8: Time needed to perform search for medium alphabet

n	m	LZ-Index	Standard FM-Index	FM-Index with Wavelet Tree
10^2	10^1	6 ms	3 ms	2 ms
10^3	10^1	36 ms	4 ms	3 ms
10^3	10^2	2.41 s	28 ms	19 ms
10^4	10^1	164 ms	14 ms	12 ms
10^4	10^2	3.53 s	39 ms	29 ms
10^5	10^2	5.64 s	76 ms	62 ms
10^6	10^2	6.21 s	82 ms	71 ms
10^6	10^3	43 s	120 ms	92 ms
10^6	10^4	392 s	124 ms	93 ms
$3 \cdot 10^6$	10^4	417 s	124 ms	94 ms

Time performance of searching in medium alphabet for both implementations of the FM-Index are quite similar, but it is much higher than for small alphabet. That is probably connected with the high memory usage. As for the LZ-Index, the time needed is around 5 times higher than in small alphabet case.

4.2.3 Alphabet of large size

Peek memory comparison

Table 4.9: Memory peak during structure creation for large alphabet

n	LZ-Index	Standard FM-Index	FM-Index with Wavelet Tree
10^2	2 MB	4.0 MB	3.0 MB
10^3	11 MB	23.0 MB	41.0 MB
10^4	33 MB	169.0 MB	406.0 MB
10^5	115.0 MB	1352.0 MB	1.1 GB
10^6	402 MB	9.32 GB	8.52 GB

Space used by the LZ-Index in that case is still much less than used by both FM-Index implementations. The interesting fact is that the FM-Index with Wavelet Tree for greater text length will use less space than the standard FM-Index implementation.

Table 4.10: Memory peak during search for large alphabet

n	m	LZ-Index	Standard FM-Index	FM-Index with Wavelet Tree
10^2	10^1	6 MB	2 KB	2 KB
10^3	10^1	8 MB	17 KB	17 KB
10^3	10^2	648 MB	15 KB	15 KB
10^4	10^1	10 MB	162 KB	162 KB
10^4	10^2	681 MB	161 KB	161 KB
10^5	10^2	822 MB	1563 KB	1614 KB
10^6	10^2	894 MB	2 MB	2 MB
10^6	10^3	10.2 GB	11.2 MB	11.1 MB

Memory performance result of searching in this case are nearly the same as the results of searching for medium alphabet. We can conclude that the size of alphabet does not have high influence on memory during search for all these structures.

Time comparison

Table 4.11: Time needed to create structures for large alphabet

n	LZ-Index	Standard FM-Index	FM-Index with Wavelet Tree
10^2	73 ms	521 ms	129 ms
10^3	324 ms	5.25 s	1916 ms
10^4	1907 ms	58.18 s	22.56 s
10^5	8.2 s	9.4 min	3.2 min
10^6	63 s	89.9 min	29.5 min

Time performance of construction these structures differs a lot in comparison to previous cases. The size of alphabet has high influence for both FM-Index implementations, but small influence for the LZ-Index. In the largest case, creation of LZ-Index is almost 30 times faster than the FM-Index.

Table 4.12: Time needed to perform search for large alphabet

n	m	LZ-Index	Standard FM-Index	FM-Index with Wavelet Tree
10^2	10^1	297 ms	10 ms	7 ms
10^3	10^1	458 ms	12 ms	14 ms
10^3	10^2	32.61 s	116 ms	98 ms
10^4	10^1	952 ms	22 ms	21 ms
10^4	10^2	38.89 s	156 ms	135 ms
10^5	10^2	41.2 s	172 ms	149 ms
10^6	10^2	44.7 s	183 ms	158 ms
10^6	10^3	7.2 min	382 ms	331 ms
10^6	10^4	81.3 min	472 ms	403 ms

In contrast to the memory performance, the time performance of that structures depends a lot on the size of the alphabet. The time needed to obtain all occurrences for LZ-Index is almost 12 times higher than for medium size alphabet.

4.2.4 Non-uniform character distribution with parameter 1/2

Peek memory comparison

Table 4.13: Memory peak during structure creation for non-uniform distribution with parameter 1/2

n	LZ-Index	Standard FM-Index	FM-Index with Wavelet Tree
10^2	101 KB	114 KB	45 KB
10^3	740 KB	897 KB	502 KB
10^4	6 MB	8 MB	5 MB
10^5	58 MB	78 MB	57 MB
10^6	534 MB	717 MB	591 MB
$3 \cdot 10^6$	5.21 GB	7.01 GB	6.05 GB

Memory performance results of construction the structures for non-uniform distribution are quite similar to the results for small alphabet. However, the results here for each structure are even more similar.

Table 4.14: Memory peak during search for non-uniform distribution with parameter 1/2

n	m	LZ-Index	Standard FM-Index	FM-Index with Wavelet Tree
10^2	10^1	5 KB	1433 B	1056 B
10^3	10^1	6 KB	1551 B	1148 B
10^3	10^2	195 KB	1620 B	1175 B
10^4	10^1	6 KB	1602 B	1249 B
10^4	10^2	198 KB	1691 B	1342 B
10^5	10^2	200 KB	1693 B	1327 B
10^6	10^2	206 KB	1694 B	1366 B
10^6	10^3	15 MB	2 KB	2 KB
10^6	10^4	1533 MB	11 KB	10 KB
$3 \cdot 10^6$	10^4	1535 MB	11 KB	11 KB

The results for the memory performance during search are exactly the same as the results of the memory performance during search for small alphabet.

Time comparison

Table 4.15: Time needed to create structures for non-uniform distribution with parameter 1/2

n	LZ-Index	Standard FM-Index	FM-Index with Wavelet Tree
10^2	3 ms	1 ms	1 ms
10^3	21 ms	19 ms	10 ms
10^4	218 ms	234 ms	210 ms
10^5	1906 ms	2.84 s	2.32 s
10^6	22.44 s	34.21 s	29.25 s
$3 \cdot 10^6$	57.2 s	98.8 s	94.3 s

The first bigger difference of the results for alphabet with non-uniform distribution and small alphabet can be seen in time performance for the structures construction. In non-uniform distribution with parameter 1/2, the creation of the both FM-Index implementations are 2 times slower comparing to LZ-Index.

Table 4.16: Time needed to perform search for non-uniform distribution with parameter 1/2

n	m	LZ-Index	Standard FM-Index	FM-Index with Wavelet Tree
10^2	10^1	1 ms	1 ms	1 ms
10^3	10^1	1 ms	1 ms	1 ms
10^3	10^2	1 ms	1 ms	1 ms
10^4	10^1	1 ms	1 ms	1 ms
10^4	10^2	2 ms	1 ms	1 ms
10^5	10^2	3 ms	1 ms	1 ms
10^6	10^2	5 ms	1 ms	1 ms
10^6	10^3	851 ms	1 ms	1 ms
10^6	10^4	79.9 s	4 ms	4 ms
$3 \cdot 10^6$	10^4	82.3 s	4 ms	4 ms

Time performance of search in this case is exactly the same as in the case with small alphabet.

4.2.5 Non-uniform character distribution with parameter 1/3

Peak memory comparison

Table 4.17: Memory peak during structure creation for non-uniform distribution with parameter 1/3

n	LZ-Index	Standard FM-Index	FM-Index with Wavelet Tree
10^2	60 KB	74 KB	34 KB
10^3	429 KB	595 KB	432 KB
10^4	3 MB	5 MB	4 MB
10^5	29 MB	48 MB	52 MB
10^6	267 MB	446 MB	536 MB
$3 \cdot 10^6$	713 MB	1281 MB	1513 MB

The results of space performance for non-uniform distribution with parameter 1/3 are quite different from for non-uniform distribution with parameter 1/2. The space needed for creation of LZ-Index is almost 2 times less than need for creation of FM-Index.

Table 4.18: Memory peak during search for non-uniform distribution with parameter 1/3

n	m	LZ-Index	Standard FM-Index	FM-Index with Wavelet Tree
10^2	10^1	6 KB	1445 B	1072 B
10^3	10^1	8 KB	1866 B	1573 B
10^3	10^2	206 KB	1668 B	1271 B
10^4	10^1	32 KB	12 KB	12 KB
10^4	10^2	216 KB	1692 B	1310 B
10^5	10^2	227 KB	1693 B	1343 B
10^6	10^2	231 KB	1694 B	1344 B
10^6	10^3	16 MB	2 KB	2 KB
10^6	10^4	1539 MB	11 KB	11 KB
$3 \cdot 10^6$	10^4	1544 MB	11 KB	11 KB

Unfortunately, the space performance during searching in this case is exactly the same as for the small alphabet case and for the non-uniform distribution with parameter 1/2, so we cannot observe any interesting properties here.

Time comparison

Table 4.19: Time needed to create structures for non-uniform distribution with parameter 1/3

n	LZ-Index	Standard FM-Index	FM-Index with Wavelet Tree
10^2	2 ms	1 ms	1 ms
10^3	13 ms	13 ms	14 ms
10^4	116 ms	137 ms	161 ms
10^5	1041 ms	1829 ms	1969 ms
10^6	10.83 s	24.37 s	26.54 s
$3 \cdot 10^6$	100.2 s	274 s	269 s

The results of time performance for construction of structures are quite similar to the results for non-uniform distribution with parameter 1/2. The only difference is that all the results here are multiplied by 2.

Table 4.20: Time needed to perform search for non-uniform distribution with parameter 1/3

n	m	LZ-Index	Standard FM-Index	FM-Index with Wavelet Tree
10^2	10^1	1 ms	1 ms	1 ms
10^3	10^1	1 ms	1 ms	1 ms
10^3	10^2	2 ms	1 ms	1 ms
10^4	10^1	3 ms	1 ms	1 ms
10^4	10^2	4 ms	1 ms	1 ms
10^5	10^2	4 ms	1 ms	1 ms
10^6	10^2	5 ms	1 ms	1 ms
10^6	10^3	844 ms	1 ms	1 ms
10^6	10^4	87.4 s	2 ms	2 ms
$3 \cdot 10^6$	10^4	92.3 s	2 ms	2 ms

As for the time performance for searching, the result does not differ at all from small alphabet and non-uniform distribution with parameter 1/2.

4.2.6 Summary

The advantage of LZ-Index is its memory usage and time needed for construction when working with large or medium alphabet. The differences in that case are huge and can reach even 25 time less memory used and 30 times less time for construction. In the other cases the results for construction are quite similar. However, time and memory needed to find occurrences for long pattern is large and may be over 100 times higher than both implementation of FM-Index. Summarizing that, LZ-Index will be better for short patterns with large alphabet.

In comparison of both implementation of FM-Index, they do not differ a lot. Time performances for searching for a pattern are exactly the same besides when the alphabet is large. However, there are difference is in its construction. Implementation based on Wavelet Tree uses less memory, but requires more time. The differences are rather small except the large alphabet when implementation based on Wavelet Tree is constructed 3 times faster. In summary, described FM-Index implementation similar in performance, but in case with large alphabet it is better to use implementation based on Wavelet Tree.

Bibliography

[KMR72] Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. “Rapid identification of repeated patterns in strings, trees and arrays”. In: *Proceedings of the 4th Annual ACM Symposium on Theory of Computing* (1972), pp. 125–136.

[KMP77] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. “Fast Pattern Matching in Strings”. In: *SIAM Journal on Computing* 6.2 (1977), pp. 323–350. DOI: 10.1137/0206024. URL: https://doi.org/10.1137/0206024.

[LZ78] A. Lempel and J. Ziv. “Compression of individual sequences via variable-rate coding”. In: *IEEE Transactions on Information Theory* 24.5 (1978), pp. 530–536. DOI: 10.1109/TIT.1978.1055934.

[Ter84] Welch A. Terry. “A Technique for High-Performance Data Compression”. In: *Computer* 17.6 (1984), pp. 8–19. DOI: 10.1109/MC.1984.1659158.

[KR87] Richard M. Karp and Michael O. Rabin. “Efficient randomized pattern-matching algorithms”. In: *IBM Journal of Research and Development* 31.2 (Mar. 1987), pp. 249–260. DOI: 10.1147/rd.312.0249. URL: https://doi.org/10.1147/rd.312.0249.

[Bae89] Ricardo Baeza-Yates. “Algorithms for string searching”. In: *ACM SIGIR Forum* 23 (Apr. 1989), pp. 34–58. DOI: 10.1145/74697.74700.

[UG90] Manber Udi and Myers Gene. “Suffix arrays: a new method for on-line string searches”. In: *First Annual ACM-SIAM Symposium on Discrete Algorithms* (1990), pp. 319–327.

[CP91] Maxime Crochemore and Dominique Perrin. “Two-way string-matching”. In: *J. ACM* 38.3 (July 1991), pp. 650–674. ISSN: 0004-5411. DOI: 10.1145/116825.116845. URL: https://doi.org/10.1145/116825.116845.

[Dav92] Ian Davis. “A Fast Radix Sort.” In: *Comput. J.* 35 (Dec. 1992), pp. 636–642. DOI: 10.1093/comjnl/35.6.636.

[Far97] Martin Farach. “Optimal suffix tree construction with large alphabets”. English (US). In: *Annual Symposium on Foundations of Computer Science - Proceedings* (1997). Proceedings of the 1997 38th IEEE Annual Symposium on Foundations of Computer Science ; Conference date: 20-10-1997 Through 22-10-1997, pp. 137–143. ISSN: 0272-5428.

[Gus97] Dan Gusfield. “Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology”. In: 1997. URL: https://api.semanticscholar.org/CorpusID:61800864.

[BF00] Michael A. Bender and Martín Farach-Colton. "The LCA Problem Revisited". In: *LATIN 2000: Theoretical Informatics*. Ed. by Gaston H. Gonnet and Alfredo Viola. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 88–94. ISBN: 978-3-540-46415-0.

[KS03] Juha Kärkkäinen and Peter Sanders. "Simple Linear Work Suffix Array Construction". In: *Automata, Languages and Programming* 2719 (2003). DOI: 10.1007/3-540-45061-0_73. URL: https://doi.org/10.1007/3-540-45061-0_73.

[Nav04] Gonzalo Navarro. "Indexing text using the Ziv–Lempel trie". In: *Journal of Discrete Algorithms* 2.1 (2004). The 9th International Symposium on String Processing and Information Retrieval, pp. 87–114. ISSN: 1570-8667. DOI: https://doi.org/10.1016/S1570-8667(03)00066-2. URL: https://www.sciencedirect.com/science/article/pii/S1570866703000662.

[FM05] Paolo Ferragina and Giovanni Manzini. "Indexing compressed text". In: *Journal of the ACM* 52.4 (2005), pp. 552–581. DOI: 10.1145/1082036.1082039. URL: https://doi.org/10.1145/1082036.1082039.

[CC07] Peter Clifford and Raphael Clifford. "Simple deterministic wildcard matching". In: *Inf. Process. Lett.* 101 (Jan. 2007), pp. 53–54. DOI: 10.1016/j.ipl.2006.08.002.

[LS07] N. Jesper Larsson and Kunihiko Sadakane. "Faster suffix sorting". In: *Theoretical Computer Science* 387.3 (2007). The Burrows-Wheeler Transform, pp. 258–272. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2007.07.017. URL: https://www.sciencedirect.com/science/article/pii/S0304397507005257.

[NZC09] Ge Nong, Sen Zhang, and Wai Hong Chan. "Linear Suffix Array Construction by Almost Pure Induced-Sorting". In: *2009 Data Compression Conference*. 2009, pp. 193–202. DOI: 10.1109/DCC.2009.42.

[NR13] Marius Nicolae and Sanguthevar Rajasekaran. "On String Matching with Mismatches". In: *Algorithms* 8 (July 2013). DOI: 10.3390/a8020248.

[Kap+14] Kumar Kapil et al. "Importance of String Matching in Real World Problems". In: 3 (July 2014), pp. 2319–7242.

[Goe15] Michel Goemans. *MIT lecture notes about LZ compressions.* Apr. 2015. URL: https://math.mit.edu/~goemans/18310S15/lempel-ziv-notes.pdf.

[CL16] Maxime Crochemore and Thierry Lecroq. *Multiple String Matching.* 2016.

[Zha19] Richard Zhan. *Segment Trees.* Lecture notes. Nov. 2019. URL: https://activities.tjhsst.edu/sct/lectures/1920/2019_11_15_Segment_Trees.pdf.

[Cor+22] T.H. Cormen et al. *Introduction to Algorithms, fourth edition.* MIT Press, 2022. ISBN: 9780262046305. URL: https://books.google.pl/books?id=HOJyzgEACAAJ.