Contents

1 Basic Datastructures

Definition 1.1
Let $\Sigma = \{0, \ldots, \sigma - 1\}$ be a finite, ordered set. The elements of Σ are called *characters* or *symbols* and Σ is called an *alphabet* of size σ.

Definition 1.2
A *string* S is a sequence of characters from an alphabet Σ.

- We usually use $n = |S|$ to be the length of the string.
- The i-th character of S is $S[i]$. Indices are 0-based.
- The substring from the i-th to the j-th character is $S[i..j]$.
- A substring with $i = 0$ is called *prefix*. A substring with $j = n - 1$ is called *suffix*.
- The i-th suffix is $S[i..n - 1]$.

1.1 Suffix Tries

Definition 1.3
Let $S = \{S_0, S_1, \ldots, S_{N-1}\}$ be a set of strings over an alphabet Σ. A *trie* $\mathcal{T}$ is a tree, where each node represents a different prefix in the set S. The root represents the empty prefix ε. Vertex u representing prefix Y is a child of vertex v representing prefix X, if and only if $Y = Xc$ for some character $c \in \Sigma$. The edge (v, u) is then labeled c.
If S is the set of all suffixes of a string T, the trie is called *suffix trie* of T.

Example 1.4
Figure 1.1 shows the suffix trie for the string "banana$". The dollar sign "$" is a sentinel that does not appear elsewhere in the text (and is lexicographically smaller than all other symbols). This guarantees, that no suffix is a prefix of another suffix and the suffix trie therefore has $n + 1$ leaves.

To construct a trie over string set $S = \{S_0, \ldots, S_{N-1}\}$, we need $\mathcal{O}(|S_0| + \ldots + |S_{N-1}|)$ steps. This bound is tight: If all characters are pairwise distinct in all strings and no two strings share a character, than the number of different prefixes and therefore vertices is given by $1 + \sum_{i=0}^{N-1} |S_i|$, where the additional 1 represents the empty prefix ε.
The time needed to search for a string T of length $m = |T|$ in the trie depends on the implementation of the tree. If the children of each vertex are stored in a list, the time is

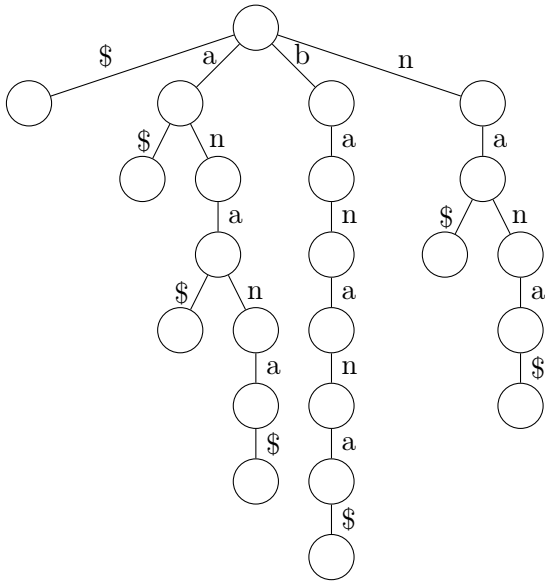

Figure 1.1: The suffix trie for the string "banana$"

in $\mathcal{O}(m\sigma)$. If the children are stored in a sorted array (using the order of the characters in the alphabet), the time is in $\mathcal{O}(m \log \sigma)$. By using a hash table and perfect hashing, the time is in $\mathcal{O}(m)$.

The space needed to store the suffix trie $\mathcal{T}$ for a string of length n is in $\mathcal{O}(n^2 \log \sigma + n^2 \log n)$ bits. The first summand is the space needed to store the $\mathcal{O}(n^2)$ edge labels of one character $c \in \Sigma$ each. The second summand is the space needed to store the pointers to the children of each node.

1.2 Suffix Trees

Definition 1.5
A *suffix tree* $\mathcal{T}$ for a string S is the suffix trie of S where each unary path is converted into a single edge. Those edges are labeled with the concatenation of the characters from the replaced edges. The leaves of the suffix tree store the text position where the corresponding suffix starts.

Example 1.6
Figure 1.2 shows the suffix tree for the string "banana$". It contains only 11 vertices compared to the 23 vertices of the suffix trie.

The suffix array can be constructed in time $\mathcal{O}(n)$ with algorithms by Weiner[4], McCreight[2] or Ukkonen[3]. It needs $\mathcal{O}(n \log n + n \log \sigma)$ bits. The first summand is the space needed for the pointers to the children and the indices stored in the leaves. The

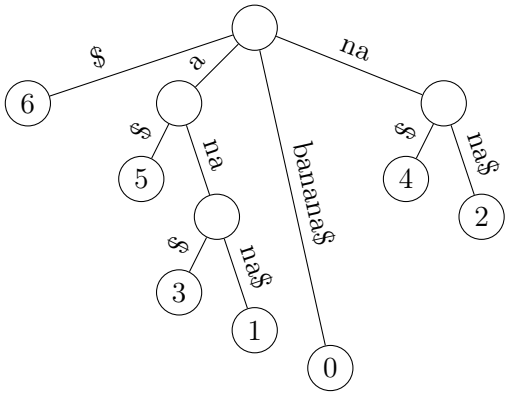

Figure 1.2: The suffix tree for the string "banana$".

second summand is the space needed for the edge labels. To achieve this space, the edge labels must not be stored explicitly. Instead we can store pointers to the first and last position of the label in the text.

In practice, a suffix tree needs more than 20 times the space of the original text. Based on the required functionality, this can even be worse.

1.3 Bitvectors

Definition 1.7
A *bitvector* B is an array of bits that are compactly stored. A bitvector supports the following operations:

- Access(i) returns the i-th element in B.
- Rank$_q(i)$ returns the number of q-bits in the prefix $[0..i-1]$ of B. If q is omitted, we assume $q = 1$.
- Select$_q(i)$ returns the position of the i-th q-bit. If q is omitted, we assume $q = 1$.

Theorem 1.8
The Rank*-Operation on bitvectors can be done constant time and $o(n)$ additional space.[1]*

Proof Let B be a bitvector of length n. Precompute the following information:

1. Divide B into *superblocks* $SB_1, \ldots, SB_{\lceil \frac{n}{L} \rceil}$ of size L.
2. For each superblock SB_j now store $\sum_{i=0}^{(j-1)L-1} B[i]$. This is the number of set bits in the first j superblocks. For each superblock this needs $\mathcal{O}(\log n)$ bits of space.
3. Now further divide each superblock B_j into blocks $B_1^j, \ldots, B^j_{\lceil \frac{L}{S} \rceil}$ of size S.

4. For each block B_k^j of superblock SB_j store $\sum_{i=(j-1)L}^{(j-1)L+kS-1} B[i]$. This is the number of set bits in the first k blocks of superblock SB_j. For each block this needs $\mathcal{O}(\log L)$ bits.

For a given RANK(i) query we now calculate the corresponding superblock SB_j and block B_k. We use the precomputed sums to get the number of set bits in all superblocks before SB_j and all blocks before B_k. All thats missing now is the number of set bits in block B_k until position i. If the blocks are small enough, this information can be precomputed efficiently for all possible blocks and positions (Four-Russians-Trick).

We still need to choose L and S:

$$L = \log^2 n \tag{1.1}$$

$$S = \frac{1}{2} \log n \tag{1.2}$$

Let's look at the space needed:

- Prefix sums among the superblocks:

$$\begin{aligned} \mathcal{O}\left(\frac{n}{L} \log n\right) &= \mathcal{O}\left(\frac{n}{\log^2 n} \log n\right) \\ &= \mathcal{O}\left(\frac{n}{\log n}\right) \end{aligned} \tag{1.3}$$

- Prefix sums among the blocks:

$$\begin{aligned} \mathcal{O}\left(\frac{n}{S} \log L\right) &= \mathcal{O}\left(\frac{n}{\log n} \log \log^2 n\right) \\ &= \mathcal{O}\left(\frac{n}{\log n} \cdot \log \log n\right) \\ &= \mathcal{O}\left(\frac{n \log \log n}{\log n}\right) \end{aligned} \tag{1.4}$$

- The lookup tables for each block. There are 2^S possible blocks and for each of them we need to store S prefix sums in $\mathcal{O}(\log S)$ bits:

$$\begin{aligned} \mathcal{O}\left(2^S \cdot S \cdot \log S\right) &= \mathcal{O}\left(2^{\frac{1}{2} \log n} \cdot \frac{1}{2} \log n \cdot \log \frac{1}{2} \log n\right) \\ &= \mathcal{O}\left(\sqrt{n} \log n \log \log n\right) \end{aligned} \tag{1.5}$$

The total amount of space needed is therefore $\mathcal{O}\left(\frac{n}{\log n} + \frac{n \log \log n}{\log n} + \sqrt{n} \log n \log \log n\right) = o(n)$ bits. ∎

1.4 Compressed Bitvector

Definition 1.9
Given a sequence X of length n over an alphabet Σ. Let n_c be the number of occurrences of character $c \in \Sigma$ in X.

$$\mathcal{H}_0(X) := \sum_{\substack{c \in \Sigma \\ n_c > 0}} \frac{n_c}{n} \log \frac{n}{n_c} \tag{1.6}$$

is called the *zeroth order empirical entropy* and provides lower bound for the number of bits needed to compress X using a compressor which just considers character frequencies.

1.4.1 Elias-Fano Encoded Bitvector

Theorem 1.10
Given a non-decreasing sequence X of length m over the alphabet $[0, n]$. Sequence X can be compressed using $2m + m \log \frac{n}{m} + o(m)$ bits while each element can still be accessed in constant time. This is known as Elias-Fano-Encoding.

Proof Divide each element into a high-part and a low-part: The first $\lfloor \log m \rfloor$ bits correspond to the high-part, the other $\lceil \log n \rceil - \lfloor \log m \rfloor$ bits correspond to the low-part. The sequence of high-parts of X is also non-decreasing. We use a unary gap encoding to represent the gaps and store the result in a bitvector H. For a gap of size δ_i we use $\delta_i + 1$ bits (δ_i zeros and 1 one). The sum of the gaps (the total number of zeros in H) is at most $2^{\lfloor \log m \rfloor} \leq 2^{\log m} = m$. Therefore H has size at most $2m$ (#zeros + #ones). The low parts can be stored explicitly.

Algorithm 1 Access to the i-th element of X.

```
Elias-Fano-Access(i)
1  p = Select_1(i + 1, H)
2  x = p − i
3  return x · 2^(⌈log n⌉−⌊log m⌋) + L[i]
```

Algorithm 1 provides a method to access the i-th element of the sequence in constant time. Variable p is the position of the i-th 1-bit (Select works 1-indexed). To get the number x of zeros until this position, we subtract i. Variable x is now giving the high part and we multiply it with $2^{\lceil \log n \rceil - \lfloor \log m \rfloor}$ to shift it in front of the low part. ∎

Theorem 1.10 can be used to compress a bitvector B. Let n be the length of B and m be the number of set bits. Further let X be the positions of the set bits. X forms an increasing sequence and Elias-Fano-Encoding can be applied.

Example 1.11
Table 1.1 shows how the bitvector $B = (4, 13, 15, 24, 26, 27, 29)$ gets encoded. The length of B is 7, so $\lfloor \log 7 \rfloor = 2$ bits are in the high-part and $\lceil \log 29 \rceil - \lfloor \log 7 \rfloor = 3$ bits are

X	=	4	13	15	24	26	27	29
		`00|100`	`01|101`	`01|111`	`11|000`	`11|010`	`11|011`	`11|101`
δ	=	0	1	0	2	0	0	0
H	=	`1011001111`						
L	=	`4, 5, 7, 0, 2, 3, 5`						

Table 1.1: Elias-Fano-Encoding for a given sequence X.

in the low-part. The second row shows the binary representation of each elements, the vertical bar separates the low- and the high-part. Line δ shows the differences between two consecutive high-parts. Lines H and L now show the compressed high-parts and the explicitly stored low-parts.

1.4.2 $\mathcal{H}_0$-Compressed Bitvector

Let B be a bitvector of length n with κ bits set. The entropy of this bitvector is

$$\mathcal{H}_0(B) = \frac{\kappa}{n} \log \frac{n}{\kappa} + \frac{n-\kappa}{n} \log \frac{n}{n-\kappa}. \tag{1.7}$$

Theorem 1.12
A bitvector can be represented in $n\mathcal{H}_0(B) + o(n)$ bits space while Rank *and* Select *queries can be executed in constant time.*

Proof Split the bitvector into blocks of length $K = \frac{1}{2} \log n$ bits. For each block i we store the number of set bits κ_i using $\lceil \log K + 1 \rceil$ bits. These identifiers sum up to $\mathcal{O}\left(\frac{n}{K} \log K\right) = \mathcal{O}\left(\frac{n \log \log n}{\log n}\right)$ bits.

Now we represent each block as a tuple (κ_i, r_i). The first element $0 \leq \kappa_i \leq K$ is the number of set bits in this block. The second element $0 \leq r_i < \binom{K}{\kappa_i}$ is the index within the class. Each value of r_i uniquely maps to one of the $\binom{K}{\kappa_i}$ possible ways a block of size K can have κ_i 1-bits.

If we sum these values for all blocks, we get the total space needed to encode B:

$$\begin{aligned}
|B| &= \sum_{i=0}^{\frac{n-1}{K}} \left\lceil \log \binom{K}{\kappa_i} \right\rceil \\
&\leq \log \left(\prod_{i=0}^{\frac{n-1}{K}} \binom{K}{\kappa_i} \right) + \left\lceil \frac{n}{K} \right\rceil \\
&\leq \log \binom{n}{\kappa_0 + \ldots + \kappa_{(n-1)/K}} + \left\lceil \frac{n}{K} \right\rceil \qquad (1.8) \\
&= \log \binom{n}{\kappa} + \left\lceil \frac{n}{K} \right\rceil \\
&\leq n\mathcal{H}_0(B) + \left\lceil \frac{n}{K} \right\rceil \\
&= n\mathcal{H}_0(B) + \mathcal{O}\left(\frac{n}{\log n} \right)
\end{aligned}$$

The first step transformed the sum of logarithms into a logarithm of a product, where the second summand originates from the ceiling function around the individual logarithms. In the second step a combinatorial argument is used: The number of ways to choose $\kappa_0, \ldots, \kappa_{\frac{n-1}{K}}$ 1-bits from all blocks is equal to the number of ways to choose $\kappa_0 + \ldots + \kappa_{\frac{n-1}{K}}$ 1-bits out of the total n bits.

In total this gives a space requirement of $n\mathcal{H}_0(B) + \mathcal{O}\left(\frac{n}{\log n}\right) + \mathcal{O}\left(n\frac{\log\log n}{\log n}\right) = n\mathcal{H}_0(B) + o(n)$ bits. We will not go into the implementation of Rank and Select here. ∎

To encode and decode a block with κ bits set we must be able to transform a bitstring into a (κ, r) pair and vice versa. One way to do this is to again use a lookup table. Since there are 2^K possible blocks, each needing $\log\lceil K+1 \rceil$ bits this would need additional $\mathcal{O}\left(\sqrt{n}\log\log n\right)$ bits.

Another way to do it on-the-fly is by using a *combinatorial number system*. The idea is to give the blocks with κ_i 1-bits consecutive numbers from 0 to $\binom{K}{\kappa_i} - 1$ in the order given by their values interpreted as binary numbers.

- Algorithm 2 shows how a given bitstring *bits* can be encoded into a pair (κ, r). Initially r is set to 0. We go through the bits from left to right. If the i-th bit is zero, we just continue with the next bit. If it is 1, we know must first number the $\binom{K-i-1}{\kappa}$ bitstrings starting with a 0, so we increase r by $\binom{K-i-1}{\kappa}$. We then decrement κ for the next iteration, because there are only $\kappa - 1$ 1-bits left for the remaining places.
- Algorithm 3 shows how to get the bitstring corresponding to a (κ, r) pair. We extract the bits consecutively from left to right. If $r \geq \binom{K-i-1}{\kappa}$, then the first bit was a 1. In this case, we decrement κ for the further places and reduce r by $\binom{K-i-1}{\kappa}$. Otherwise the first place was a 0.

Algorithm 2 Encodes bitstring *bits* of length K using a combinatorial number system.

```
EncodeBlock(bits)
1  r = 0
2  κ = CountSetBits(bits)
3  for i = 0 to K
4       if bits[i] == 1
5            r = r + binom(K-i-1, κ)
6            κ = κ - 1
7  return (κ, r)
```

Algorithm 3 Decodes a (κ, r) pair to a bitstring of length K.

```
DecodeBlock(κ, r)
1  bits = ε
2  for i = 0 to K
3       if r ≥ binom(K-i-1, κ)
4            bits = bits + 1
5            r = r - binom(K-i-1, κ)
6       else
7            bits = bits + 0
8  return bits
```

All the binomial coefficients needed in Algorithms 2 and 3 need to be precomputed to allow $O(1)$ access.

Example 1.13
Assume $K = 6$ and we are given a block *bits* = `100110`. It contains three 1-bits, so $\kappa = 1$. We will now find r:

1. Initialize $r = 0$.
2. The first bit is 1, so increase r by $\binom{6-0-1}{3} = 10$.
3. The second and third bits are 0, so we skip them.
4. The fourth and the fifth bit are 1, we increase r by $\binom{6-3-1}{2} = 1$ and by $\binom{6-4-1}{1} = 1$, so r becomes 12.

We see that Algorithm 2 mapped `100110` $\mapsto (3, 12)$. To check our calculations we will now decode this value:

1. $12 \geq \binom{6-0-1}{3} = 10$, so the first bit is a 1 and we reduce r by 10 to 2.
2. $2 < \binom{6-1-1}{2} = 6$ and $2 < \binom{6-2-1}{2} = 3$, so the second and third bit are both 0.
3. $2 \geq \binom{6-3-1}{2} = 1$, so the fourth bit is a 1 and we reduce r by 1 to 1.

4. $1 \geq \binom{6-4-1}{1} = 1$, so the fifth bit is again a 1 an we further reduce r by 1 to 0.
5. $0 < \binom{6-5-1}{0} = 0$, so the sixth bit is 0.

We see, that Algorithm 3 correctly mapped $(3, 12) \mapsto$ `100110`.

1.5 Wavelet Trees

Definition 1.14
A *wavelet tree* is a compact datastructure that stores a sequence S and generalizes the operations of a bitvector to an arbitrary alphabet.

- Access(i) returns the i-th element of the sequence.
- Rank$_q(i)$ returns the number of occurrences of q in the prefix $S[0..i-1]$.
- Select$_q(i)$ returns the position of the i-th occurrence of q in S.

The root of the wavelet tree stores the whole sequence. Each vertex recursively divides its sequence to its two children. The left child contains the first half of the remaining alphabet, the right child contains the second half of the remaining alphabet. A bitvector in every vertex stores the corresponding child for each element.

Lemma 1.15
A wavelet tree can be stored in $n\lceil \log \sigma \rceil$ bits space.

Proof The wavelet tree has height $\lceil \log \sigma \rceil$ and stores n bits on every layer (maybe even less on the last layer). Therefore $n\lceil \log \sigma \rceil$ bits are needed to store the bitvectors. A wavelet tree can be implemented fully via bitvectors and does not need any pointers. This will be demonstrated in more detail below. ∎

Example 1.16
Figure 1.3 shows the wavelet tree for the string "abracadabra". By concatenating the bitvectors in each vertex in level order from left to right, we fully describe the wavelet tree. The bitvector describing this wavelet tree is `00100010010|00010000|101|0100010`, where the vertical lines show the borders between consecutive vertices. They do not need to be stored, because all layers (except maybe the last layer) contain exactly n bits.

When storing the wavelet tree in a single bitvector B as in Example 1.16, each vertex can be described by two indices giving the position of the bitvector of the vertex in B. For example the root corresponds to the pair $[0, n-1]$. Algorithm 4 shows how to get the level of a vertex v of the wavelet tree. This just uses the fact, that every level contains n bits. Algorithm 5 calculates the length of the string stored in the given vertex. Algorithm 6 and Algorithm 7 return the indices for the left and the right child of the given vertex. Assuming that we can execute the rank queries in constant time (Theorem 1.8), they both run in constant time. It is also possible to implement a Parent-function, but this

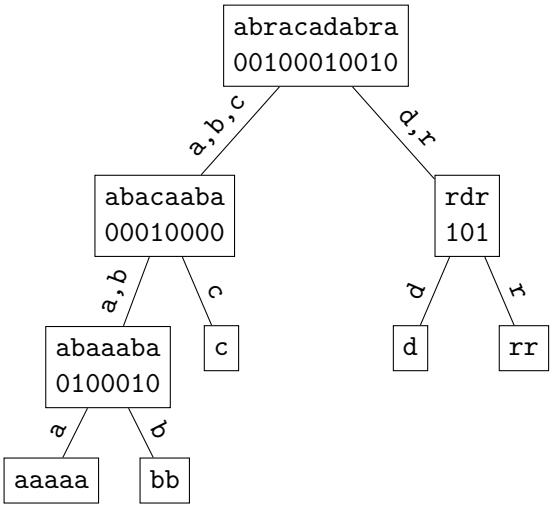

Figure 1.3: The wavelet tree for the string "abracadabra".

needs additional information, such as whether the current vertex is a left or a right child of its parent.

Algorithm 4 Returns the level of a vertex of the wavelet tree.

LEVEL($v = [l, r]$)
1 **return** $\lceil \frac{l}{n} \rceil$

Algorithm 5 Returns the number of elements stored in a vertex of the wavelet tree.

SIZE($v = [l, r]$)
1 **return** $r - l + 1$

Theorem 1.17
The ACCESS(i)*-operation can be implemented in* $\mathcal{O}(\log \sigma)$.

PROOF To access the i-th element we check the i-th position in the root-bitvector. If it is 0, the element is stored in the left child and the new index there is $i - \text{RANK}(i)$. If it is 1, the element is in the right child and the new index there is $i - \text{RANK}_0(i)$, where $\text{RANK}_0(i) := i - \text{RANK}(i)$ is the number of 0-bits before position i.

This can be done in $\mathcal{O}(\log \sigma)$, because the wavelet tree has a height of $\log \sigma$ and the rank queries can be done in constant time on bitvectors. ∎

Theorem 1.18
The $\text{RANK}_q(i)$*-operation can be implemented in* $\mathcal{O}(\log \sigma)$.

PROOF $\text{RANK}_q(i)$-queries can be answered the same way as ACCESS(i)-queries. The ACCESS(i)-query descents into the leaf containing all q symbols with some modified

Algorithm 6 Returns the left child of a vertex of the wavelet tree.

LEFT-CHILD($v = [l, r]$)
1 $l' = l + n$
2 $r' = l' + \text{SIZE}(v) - (\text{RANK}(r + 1) - \text{RANK}(l)) - 1$
3 **return** $[l', r']$

Algorithm 7 Returns the right child of a vertex of the wavelet tree.

RIGHT-CHILD($v = [l, r]$)
1 $r' = r + n$
2 $l' = r' - (\text{RANK}(r + 1) - \text{RANK}(l) - 1)$
3 **return** $[l', r']$

index i'. The rank is the number of elements before i', so $i' - 1$. Since no additional work needs to be done, the runtime is in $\mathcal{O}(\log \sigma)$. ∎

Theorem 1.19
The $\text{SELECT}_q(i)$*-operation can be implemented in* $\mathcal{O}(\log \sigma)$.

PROOF For a $\text{SELECT}_q(i)$-query we start in the leave corresponding to symbol q at position i. This can be found the same way as in the ACCESS-operation. Now we recursively process the parents until reaching the root. If the current vertex is a left child, the new position is the position of the i-th 0 in the parent bitvector. If it is a right child, the new position is the position of the i-th 1 in the parent bitvector. This needs SELECT-queries on bitvectors, which can be done in constant time (not part of this document yet). ∎

1.6 $\mathcal{H}_0$-Compression for Sequences

Definition 1.20
Let S be a sequence of length n over an alphabet $\Sigma = [0, \sigma - 1]$ of size σ. Again n_c is the number of occurrences of $c \in \Sigma$ in S.

$$\mathcal{H}_0(S) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c} \tag{1.9}$$

The idea to compress S is to represent S via a bitvector using compressed bitvectors.

2 Suffix Arrays

2.1 Suffix Array

Definition 2.1
The *suffix array* SA of a string T gives the sorted order of all suffixes of T. Element $SA[i]$ stores the index of the i-th lexicographically smallest suffix of T.

Example 2.2
Consider the string $T =$ `abracadabrabarbara$`. Table 2.1 shows the suffix array for T. Note that the dollar sign `$` is lexicographically smaller than any other character. This guarantees that suffixes which are a prefix of other suffixes appear first.

i	$SA[i]$	$T[SA[i]..n-1]$
0	18	`$`
1	17	`a$`
2	10	`abarbara$`
3	7	`abrabarbara$`
4	0	`abracadabrabarbara$`
5	3	`acadabrabarbara$`
6	5	`adabrabarbara$`
7	15	`ara$`
8	12	`arbara$`
9	14	`bara$`
10	11	`barbara$`
11	8	`brabarbara$`
12	1	`bracadabrabarbara$`
13	4	`cadabrabarbara$`
14	6	`dabrabarbara$`
15	16	`ra$`
16	9	`rabarbara$`
17	2	`racadabrabarbara$`
18	13	`rbara$`

Table 2.1: The suffix array for `abracadabrabarbara$`.

Theorem 2.3
We can search for all occurrences of a string S in T using the suffix array SA over T in time $\mathcal{O}(m \log n)$, where $n = |T|$ and $m = |S|$. This is knows as forward search.

Proof We find the first suffix in SA greater or equal to S and the first suffix in T bigger than S. Assume these are at positions i and j with $i \leq j$. Then we have $j - i$ occurrences of S in T and the corresponding positions are stored in $SA[i], \ldots, SA[j-1]$. Both i and j can be found using a binary search, since the suffix array is sorted. The binary search needs $\mathcal{O}(\log n)$ steps and each steps does a string comparison in $\mathcal{O}(m)$ time. ∎

Example 2.4
Consider strings $T =$ `abracadabrabarbara$` and $S =$ `bar`. How often and where does S appear in T?

Table 2.1 shows the suffix array for T. By binary search we find $i = 9$ and $j = 11$. So $T[SA[i]\,..\,n-1]$ is the smallest suffix greater or equal than `bar` and $T[SA[j]\,..\,n-1]$ is the smallest suffix greater than `bar`. We see that `bar` appears $j - i = 11 - 9 = 2$ times. The occurrences are at positions $SA[9] = 14$ and $SA[10] = 11$.

The suffix array itself needs $n \log n$ bits of space. But to work with it, we additionally need to store the text itself, which needs another $n \log \sigma$ bits of space. In total this results in a space of $n \log n + n \log \sigma$ bits.

2.2 Burrows-Wheeler-Transformation

Definition 2.5
The *Burrows-Wheeler-Transformation* BWT rearranges the characters of a string T into runs of similar characters which makes it easier to compress.

$$BWT[i] := T[SA[i] - 1 \mod n] \tag{2.1}$$

Intuitively, $BWT[i]$ is the character preceding the i-th suffix (or the last character for the zeroth suffix).

The BWT needs $n \log \sigma$ bits of space if it is stored uncompressed. The idea behind the BWT was better compression: Compressed size is $nH_k(T)$ bits, where $H_k(T)$ is the $k-th$ order entropy of the text T.

Example 2.6
Table 2.2 shows the suffix array for the string `abracadabrabarbara$` together with the Burrows-Wheeler-Transformation of it.

Theorem 2.7
We can search for all occurrences of a string S in T using the suffix array SA and the Burrwos-Wheeler-Transform BWT over T in time $\mathcal{O}(m \log \sigma)$, where $m = |S|$. This is knows as backward search.

i	$SA[i]$	$BWT[i]$	$T[SA[i]\,..\,n-1]$
0	18	a	$
1	17	r	a$
2	10	r	abarbara$
3	7	d	abrabarbara$
4	0	$	abracadabrabarbara$
5	3	r	acadabrabarbara$
6	5	c	adabrabarbara$
7	15	b	ara$
8	12	b	arbara$
9	14	r	bara$
10	11	a	barbara$
11	8	a	brabarbara$
12	1	a	bracadabrabarbara$
13	4	a	cadabrabarbara$
14	6	a	dabrabarbara$
15	16	a	ra$
16	9	b	rabarbara$
17	2	b	racadabrabarbara$
18	13	a	rbara$

Table 2.2: The suffix array for abracadabrabarbara$.

Proof We need another array C storing for each $c \in \Sigma$ the position of the first suffix in SA starting with c. The numbers in C are sorted the same way as the characters in Σ. Array C needs another $\sigma \log n$ bits of space.

We search for all suffixes of T starting with S. They form a consecutive interval in the suffix array SA of T. We start with the full interval $[sp_0, ep_0] = [0, n-1]$ corresponding to all suffixes of T starting with the empty suffix ε of S. In step i ($1 \leq i \leq m$), we will shrink the interval to $[sp_i, ep_i]$ corresponding to all suffixes of T starting with the last i characters of S (this is why its called backward search). The new interval $[sp_{i+1}, ep_{i+1}]$ is defined as

$$sp_{i+1} = C[c] + \textsc{Rank}_c(sp_i, BWT) \tag{2.2}$$
$$ep_{i+1} = C[c] + \textsc{Rank}_c(ep_i + 1, BWT) - 1 \tag{2.3}$$

where c is the $(i+1)$-th last character of S. Both operations can be done in $\mathcal{O}(\log \sigma)$ using a wavelet tree over the BWT array. The search needs m steps, so the total runtime is in $\mathcal{O}(m \log \sigma)$.

The interval given by $[sp_m, ep_m]$ in the suffix array corresponds to all occurrences of S in T and the positions are $SA[sp_m], \ldots, SA[ep_m]$. ∎

The intuition behind Equation 2.2 and 2.3 is the following: $C[c]$ gives the start position of all suffixes in SA starting with the $(i+1)$-th character c of S. But of course, only some continuous subsequence of them also starts with the whole last $i+1$ characters of S. This subsequence is calculated with the RANK_c queries: The BWT tells us, which of the suffixes in $[sp_i, ep_i]$ are preceded with c (the $(i+1)$-th last character of S). To now calculate sp_{i+1} from i, we need to know, how many suffixes starting with c are lexicographically smaller than the c-preceded suffixes in $[sp_i, ep_i]$. The answer is just $\text{RANK}_c(sp_i, BWT)$ as in Equation 2.2. For ep_{i+1} Equation 2.3 additionally counts the number of suffixes of T in $[sp_i, ep_i]$ preceded by c.

Example 2.8
We will again search for string $S =$ `bar` in the text $T =$ `abracadabrabarbara$`, this time using backward search. The suffix array and Burrows-Wheeler-Transformation of T is given in Table 2.2.

Array C is given by the following Table 2.3 for text T.

$	a	b	c	d	r
0	1	9	13	14	15

Table 2.3: C-array for the string `abracadabrabarbara$`.

Initially $[sp_0, ep_0] = [0, n-1]$ is the whole set of all suffixes of T. They all match the empty suffix ε of S. We will now go through the different steps of the backward search, the red part always highlights the current suffix of S matched with the suffixes in SA.

1. Step 1 (`bar`):

$$\begin{aligned} sp_1 &= C[r] + \text{RANK}_r(sp_0, BWT) \\ &= 15 + \text{RANK}_r(0, BWT) \\ &= 15 + 0 = 15 \\ ep_1 &= C[r] + \text{RANK}_r(ep_0 + 1, BWT) - 1 \\ &= 15 + \text{RANK}_r(19, BWT) \\ &= 15 + 4 - 1 = 18 \end{aligned}$$

   The suffixes starting with `r` start at position $C[r] = 15$ in the suffix array and there is a total of four of them, so $[sp_1, ep_1] = [15, 18]$.

2. Step 2 (`bar`):

$$\begin{aligned} sp_2 &= C[a] + \text{RANK}_a(sp_1, BWT) \\ &= 1 + \text{RANK}_a(15, BWT) \\ &= 1 + 6 = 7 \\ ep_2 &= C[a] + \text{RANK}_a(ep_1 + 1, BWT) - 1 \\ &= 1 + \text{RANK}_a(19, BWT) \\ &= 1 + 8 - 1 = 8 \end{aligned}$$

Of the four suffixes starting with `r` found in step 1, two are preceded by `a` (at position 15 and 18 in the suffix array). Further there is $\text{RANK}_a(15, BWT) = 6$ more suffixes preceded by `a` not starting with `r` that are lexicographically smaller, so the new interval becomes $[sp_2, ep_2] = [7, 8]$.

3. Step 3 (`bar`):

$$\begin{aligned} sp_3 &= C[b] + \text{RANK}_b(sp_2, BWT) \\ &= 9 + \text{RANK}_b(7, BWT) \\ &= 9 + 0 = 9 \\ ep_3 &= C[b] + \text{RANK}_b(ep_2 + 1, BWT) - 1 \\ &= 9 + \text{RANK}_b(8, BWT) \\ &= 9 + 2 - 1 = 10 \end{aligned}$$

Both suffixes starting with `ar` in SA are preceded with `b` and there is no other, smaller suffixes in SA preceded by `b`, so the two suffixes starting with `bar` are given by $[sp_3, ep_3] = [9, 10]$.

2.3 Self Index

The suffix array together with the Burrows-Wheeler-Transformation allowed to count the occurrences of a pattern S in a string T in $\mathcal{O}(m \log \sigma)$. The original text was not needed at all. But when we want to print the actual occurrences, we need to access the text T and therefore store it together with the index. We will now see, how we can code the text into the index.

Definition 2.9
A *self index* is an index that allows fast pattern matching and efficient reconstruction of any substring of the original text.

Definition 2.10
Let $j = SA[i]$ be the starting position of the i-th smallest suffix. Then $LF[i]$ is defined as the position of $(j - 1) \mod n$ in the suffix array.

Theorem 2.11
LF can be calculated from the Burrows-Wheeler-Transformation:

$$LF[i] := C[BWT[i]] + \text{RANK}_{BWT[i]}(i, BWT) \tag{2.4}$$

PROOF We know that $BWT[i]$ is the character preceding the suffix at position i in the suffix array. So the previous suffix must be in the continuous range of suffixes in the suffix array starting with $BWT[i]$. This range begins at index $C[BWT[i]]$. Then $\text{RANK}_{BWT[i]}(i, BWT)$ calculates the offset in this range by just counting how many suffixes also start with $BWT[i]$ and are lexicographically smaller. ∎

Theorem 2.12
The Burrows-Wheeler-Transformation and the LF-array are enough information to decode the whole text T.

PROOF We will proof this by induction.

Base: We know that the last suffix is the dollar sign `$` and that it is stored at index 0 in the suffix array SA, because `$` is lexicographically smaller than all other characters of our alphabet.

Step: Assume we know some character c and the index i, where the suffix of our text starting at c is in the suffix array. Then the Burrows-Wheeler-Transformation $BWT[i]$ already gives us the character preceding c. To be able to pull of the same trick again, we still need the position of the suffix preceding the one starting at c. This is just how $LF[i]$ was defined. ∎

Definition 2.13
The *F-array* contains the first characters of each suffix of text T in the order they appear in the suffix array.

$$F[i] := T[SA[i]] \tag{2.5}$$

Definition 2.14
The inverse of LF is called Ψ and maps $j = SA[i]$, the position of the i-th smallest suffix, to the position of $(j + 1) \mod n$ in the suffix array.

Theorem 2.15
For a text T we get

$$\Psi[i] := \text{SELECT}_{F[i]}(\text{RANK}_{F[i]}(i, F), BWT). \tag{2.6}$$

PROOF If $j = SA[i]$ is the position of the i-th lexicographically smallest suffix, then the suffix starting at position $j + 1$ is preceded by $F[i]$. Therefore we know the Burrows-Wheeler-Transformation $BWT[\Psi[i]] = F[i]$. This is why we can calculate Ψ by a SELECT-query on the BWT-array. The inner query $\text{RANK}_{F[i]}(i, F)$ determines which occurrence of $F[i]$ in the BWT-array we are interested in. If the suffix at position i in the suffix array is the k-th smallest starting with $F[i]$, then we look for the k-th occurrence of $F[i]$ in the BWT-array. ∎

Function Ψ allows to reconstruct the text from the front in contrast to LF, which can reconstruct it from the back as described in Theorem 2.12.

Definition 2.16
The suffix array is a permutation of the integers in $[0, n]$. The inverse permutation ISA is called the *inverse suffix array*. We get $i = ISA[SA[i]]$.

Theorem 2.17
LF and Ψ can be expressed using only the suffix array SA and the inverse suffix array ISA.

$$LF[i] = ISA[(SA[i] - 1) \mod n] \tag{2.7}$$
$$\Psi[i] = ISA[(SA[i] + 1) \mod n] \tag{2.8}$$

PROOF $LF[i]$ was defined as the position of $(SA[i] - 1) \mod n$ in the suffix array. If we know the inverse permutation of the suffix array ISA, we can just look into it for the position we want. For Ψ this works analogously. ∎

Theorem 2.18
SA, ISA, LF and Ψ can be accessed using $n \log n + o(n \log n)$ bits space each in time $\mathcal{O}(\log n)$.

PROOF We can build a wavelet tree over the suffix array SA. By Theorem 1.17 we can than access each element in the suffix array in time $\mathcal{O}(\log n)$. To access the inverse suffix array ISA we would need the position of some value i in the suffix array. This can be done with a single SELECT-query on the wavelet tree. By Theorem 1.19 this takes time $\mathcal{O}(\log n)$ time as well. Last, we know from Theorem 2.17 that efficient access to SA and ISA is enough to calculate LF and Ψ. ∎

Example 2.19
Table 2.4 shows the suffix array for the string $T =$ `abracadabrabarbara$` together with all the different arrays introduced in this section.

Theorem 2.20
The values of Ψ form at most σ increasing integer sequences. Consider Table 2.4 as an example.

PROOF We can divide the entries in the suffix array into σ continuous parts, grouping the suffixes starting with the same symbol $c \in \Sigma$. In each group the suffixes are sorted lexicographically. Because the first character in each group is the same for each suffix, their second characters form an increasing sequence. Ψ maps to the positions of the suffixes starting at these second characters, so the Ψ values are also increasing. ∎

2.4 Sampling

Instead of storing the complete suffix array we can only store a subset of the values to save space.

i	$SA[i]$	$LF[i]$	$\Psi[i]$	$BWT[i]$	$F[i]$	$T[SA[i]\,..\,n-1]$
0	18	1	4	a	$	$
1	17	15	0	r	a	a$
2	10	16	10	r	a	abarbara$
3	7	14	11	d	a	abrabarbara$
4	0	0	12	$	a	abracadabrabarbara$
5	3	17	13	r	a	acadabrabarbara$
6	5	13	14	c	a	adabrabarbara$
7	15	9	15	b	a	ara$
8	12	10	18	b	a	arbara$
9	14	18	7	r	b	bara$
10	11	2	8	a	b	barbara$
11	8	3	16	a	b	brabarbara$
12	1	4	17	a	b	bracadabrabarbara$
13	4	5	6	a	c	cadabrabarbara$
14	6	6	3	a	d	dabrabarbara$
15	16	7	1	a	r	ra$
16	9	11	2	b	r	rabarbara$
17	2	12	5	b	r	racadabrabarbara$
18	13	8	9	a	r	rbara$

Table 2.4: The suffix array for $T =$ abracadabrabarbara$.

Definition 2.21
Fix a sampling parameter s and add a bitvector B of length n with $B[i] = 1$ if and only if $SA[i] \equiv 0 \mod s$. Store exactly those elements of SA in another array SA' of size $\lceil \frac{n}{s} \rceil$, the *sampled suffix array*. For all i with $B[i] = 1$, we get

$$SA'[\text{Rank}_1(i, B)] := SA[i]. \tag{2.9}$$

This is known as *SA-order-sampling*.

Definition 2.22
Fix a sampling parameter s. Store exactly those elements $SA[i]$ with $i \equiv 0 \mod s$ in another array SA' of size $\lceil \frac{n}{s} \rceil$. We get

$$SA'[i/s] := SA[i]. \tag{2.10}$$

This is known as *text-order-sampling*.

No matter which of the above sampling strategies above is used, the code in Algorithm 8 allows to access arbitrary elements of the suffix array.

Theorem 2.23
If SA-order-sampling is used, the Access-Sampled-Suffix-Array*-method in Algorithm 8 takes at most $\mathcal{O}(s \cdot t_{LF})$ time, where t_{LF} is the time for an access in the LF array.*

Algorithm 8 Access the sampled suffix array.

ACCESS-SAMPLED-SUFFIX-ARRAY(i)

```
1  k = 0
2  while B[i] == 0
3        k = k + 1
4        i = LF[i]
5  return SA'[RANK_1(i, B)] + k
```

PROOF Assume we want to access index i of the suffix array SA and let $j = SA[i]$. Each $i = LF[i]$ call executed in Algorithm 8 j becomes $j - 1$, so after at most $s - 1$ iterations $j - k = SA[i]$ is divisible by s and therefore sampled. We can return $j - k + k = j$. ∎

Theorem 2.24
SA-order-sampling as described needs at most $\frac{n}{s} \log n + 2n$ *bits space. This can be improved to* $\frac{n}{s} \log \frac{n}{s} + 2n$ *bits, with the* ACCESS-SAMPLED-SUFFIX-ARRAY*-operation still needing time in* $\mathcal{O}(s \cdot t_{LF})$.

PROOF The sampled suffix array has $\frac{n}{s}$ elements, each an integer from $[0, n-1]$. The additional $2n$ bits are n-bit bitvector B and the overhead to allow RANK-operations in constant time.

The necessary observation to reduce the space needed is that each sampled element $SA[i]$ is a multiple of the sampling parameter s. So instead of

$$SA'[\text{RANK}_1(i, B)] := SA[i] \tag{2.11}$$

we can write

$$SA'[\text{RANK}_1(i, B)] := \frac{SA[i]}{s}. \tag{2.12}$$

The access operation stays the same, instead of the return call in the last line: We need to multiply the value read from SA' by s. ∎

Theorem 2.25
The inverse suffix array ISA can be sampled in $\frac{n}{s} \log n$ *bits space using an array with* $\frac{n}{s}$ *elements and allowing to access arbitrary elements of ISA in time* $\mathcal{O}(s \cdot t_{LF})$.

PROOF $ISA[i]$ gives the lexicographical rank of the i-th suffix. Therefore

$$LF[ISA[i]] = ISA[(i-1) \mod n]. \tag{2.13}$$

By induction we get

$$LF^k[ISA[i]] = ISA[(i-k) \mod n]. \tag{2.14}$$

We now choose a sampling parameter s and allocate an array ISA' with $\frac{n}{s}$ elements. For every $i \equiv 0 \mod s$ we store

$$ISA'[i/s] = ISA[i]. \tag{2.15}$$

To access $ISA[i]$ for some i we can use the code from Algorithm 9.

Algorithm 9 Access the sampled inverse suffix array.

ACCESS-SAMPLED-INVERSE-SUFFIX-ARRAY(i)

1 $idx = \lceil \frac{i}{s} \rceil$
2 $diff = idx - i$
3 **return** $LF^{diff}[ISA'[idx]]$ ■

The bitvector storing the sampled positions of the the suffix array when using SA-order-sampling is a considerable space overhead for texts over a small alphabet. This overhead is saved when using text-order-sampling. However the upper bound of at most $s - 1$ LF-calls does not hold any more. Consider a text like `abcdeabcde$` that consists of two copies of the same string and a sampling parameter of $s = 2$. Table 2.5 shows the corresponding suffix array. The stars indicate which elements are sampled ($SA[i] \equiv 0$ for SA-order-sampling, $i \equiv 0$ for text-order-sampling).

i	$SA[i]$	$SA[i] \equiv 0$	$i \equiv 0$	$T[i \,..\, n-1]$
0	10	*	*	`$`
1	5			`abcde$`
2	0	*	*	`abcdeabcde$`
3	6	*		`bcde$`
4	1		*	`bcdeabcde$`
5	7			`cde$`
6	2	*	*	`cdeabcde$`
7	8	*		`de$`
8	3		*	`deabcde$`
9	9			`e$`
10	4	*	*	`eabcde$`

Table 2.5: The suffix array of $T =$ `abcdeabcde$` with different sampling positions. The sampling parameter is $s = 2$.

Imagine we want to access $SA[9]$. Both sampling strategies did not sample this value. SA-order-sampling needs at most $s - 1 = 1$ LF-call. Text-order-sampling however needs 5 LF-calls. In general for a text like this of length n, $\frac{n}{s}$ LF-calls are necessary in the worst case, which is linear in the text length.

2.5 Compressed Suffix Array

Theorem 2.20 tells us that the values of Ψ form at most σ increasing sequences in the range $[0, n-1]$. We can use this fact to compress this data and obtain what is called a Ψ-based *compressed suffix array* CSA_Ψ.

2.5.1 Elias-Fano-Encoded CSA_Ψ

We can use Elias-Fano-Encoding (Theorem 1.10) to encode each of the increasing subsequences of Ψ. Let's calculate the needed space for each of these sequences. In the following equation the alphabet Σ is the set of possible gap lengths. For each $c \in \Sigma$, n_c is the number of occurrences of c as a gap in the increasing sequence.

$$\begin{aligned} |CSA_\Psi| &\overset{1.10}{=} \sum_{c\in\Sigma} \left(2n_c + n_c \log \frac{n}{n_c} + o(n_c)\right) \\ &= \sum_{c\in\Sigma} 2n_c + n \sum_{c\in\Sigma} \frac{n_c}{n} \log \frac{n}{n_c} + o(n) \\ &\overset{1.9}{=} 2n + n\mathcal{H}_0(T) + o(n) \end{aligned} \tag{2.16}$$

In addition to these $2n + n\mathcal{H}_0(T) + o(n)$ bits we need another $\mathcal{O}(\sigma \log n)$ bits to store the character boundaries.

Algorithm 10 Compare a pattern P to suffix $SA[i]$.

```
COMPARE-CSA(P, i)
1  k = 0
2  while k < |P|
3      if C[P[k] + 1] − 1 < i
4          return −1          // P smaller than suffix SA[i].
5      elseif C[P[k]] > i
6          return 1           // P larger than suffix SA[i].
7      k = k + 1
8      i = Ψ[i]
9  return 0                   // P equal to the first |P| character of the suffix SA[i].
```

We can use Algorithm 10 to search for some pattern P in the compressed suffix array. All we need to store is array C giving the first position for each character and array Ψ. We can then binary search for the correct position each time using COMPARE-CSA to check whether the corresponding entry matches P. COMPARE-CSA compares the pattern from left to right. Index $C[P[k]+1]-1$ in Line 3 is the last position in the suffix array starting with $P[k]$ while index $C[P[k]]$ is the first position in the suffix array starting with $P[k]$. The whole process needs time $\mathcal{O}(m \log n \log \sigma)$.

2.5.2 Elias-δ-Encoded CSA_Ψ

Another way to encode and compress the values of Ψ is by using an Elias-δ-code.

Definition 2.26
Elias-δ-Encoding is a way to represent a positive integer x using $\lfloor \log(x) \rfloor + 2\lfloor (\lfloor \log(x) \rfloor + 1) \rfloor + 1$ bits.

To encode x execute the following steps:

1. Let $N = \lfloor \log x \rfloor$ be the highest power of two in x, so $2^N \leq x < 2^{N+1}$.
2. Let $L = \lfloor \log N + 1 \rfloor$ be the highest power of two in $N+1$, so $2^L \leq N+1 < 2^{L+1}$.
3. Write L zeros.
4. Write the $L+1$ bit representation of $N+1$.
5. Write the last N bits of X (all but the leading one).

Let's now calculate the space needed to store the values of Ψ if an Elias-δ-Encoding is used. Again, the set of possible gaps is the alphabet and n_c is the number of occurrences of a symbol (i.e. gap) in the values of Ψ. Furhter we define

$$g_{c,i} = \begin{cases} \Psi[C[c]] & \text{for } i = 0 \\ \Psi[C[c]+i] - \Psi[C[c]+i-1] & \text{for } i > 0 \end{cases} \tag{2.17}$$

to be the i-th gap length of the increasing sequence corresponding to symbol c.

$$\begin{aligned} |CSA_\Psi| &\overset{2.26}{=} \sum_{c\in\Sigma} \sum_{i=0}^{n_c-1} (\log g_{c,i} + 2\log\log g_{c,i} + \mathcal{O}(1)) \\ &\leq \mathcal{O}(n) + \sum_{c\in\Sigma} \sum_{i=0}^{n_c-1} \left(\log \frac{n}{n_c} + 2\log\log \frac{n}{n_c} \right) \\ &= \mathcal{O}(n) + n \sum_{c\in\Sigma} \frac{n_c}{n} \left(\log \frac{n}{n_c} + 2\log\log \frac{n}{n_c} \right) \\ &= n\mathcal{H}_0(T) + \mathcal{O}(n \log\log n) \end{aligned} \tag{2.18}$$

In the first step we just applied the definition of Elias-δ-Encoding, before estimating an upper bound for the logarithms in the second step. In the third step we now used that the summation index i does not appear in the expression anymore. The we split the summation into two sums and plug the definition for the zeroth-order entropy.

Index

Bibliography

[1] G. Jacobson. "Space-efficient static trees and graphs". In: *30th Annual Symposium on Foundations of Computer Science*. Oct. 1989, pp. 549–554. DOI: 10.1109/SFCS.1989.63533.

[2] Edward M. McCreight. "A Space-Economical Suffix Tree Construction Algorithm". In: *J. ACM* 23.2 (Apr. 1976), pp. 262–272. ISSN: 0004-5411. DOI: 10.1145/321941.321946. URL: http://doi.acm.org/10.1145/321941.321946.

[3] E. Ukkonen. "On-line construction of suffix trees". In: *Algorithmica* 14.3 (Sept. 1995), pp. 249–260. ISSN: 1432-0541. DOI: 10.1007/BF01206331. URL: https://doi.org/10.1007/BF01206331.

[4] Peter Weiner. "Linear Pattern Matching Algorithms". In: *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973)*. SWAT '73. Washington, DC, USA: IEEE Computer Society, 1973, pp. 1–11. DOI: 10.1109/SWAT.1973.13. URL: https://doi.org/10.1109/SWAT.1973.13.