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

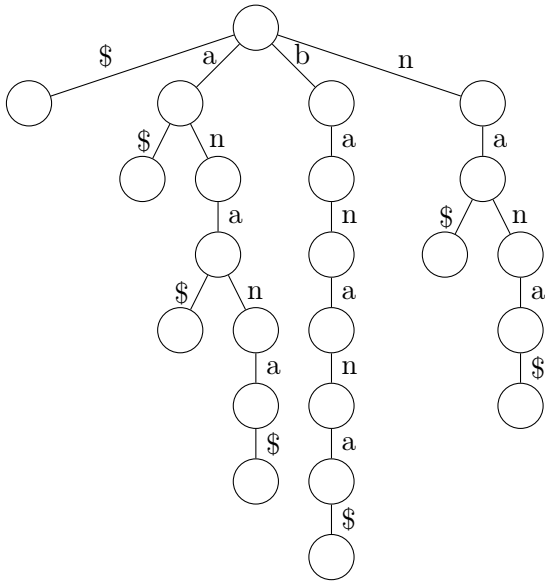

Figure 1.1: The suffix trie for the string "banana$".

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[11], McCreight[6] or Ukkonen[10]. 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.

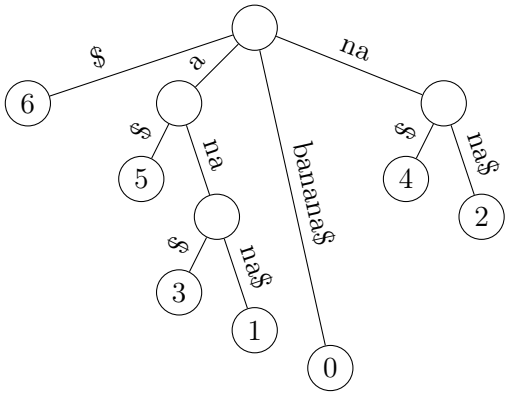

Figure 1.2: The suffix tree for the string "banana$".

The 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:

- $\textsc{Access}(i)$ returns the i-th element in B.
- $\textsc{Rank}_q(i)$ returns the number of q-bits in the prefix $[0..i-1]$ of B. If q is omitted, we assume $q = 1$.
- $\textsc{Select}_q(i)$ 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.[5]*

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 $B_1^j, \ldots, B_{\lceil \frac{L}{S} \rceil}^j$ 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

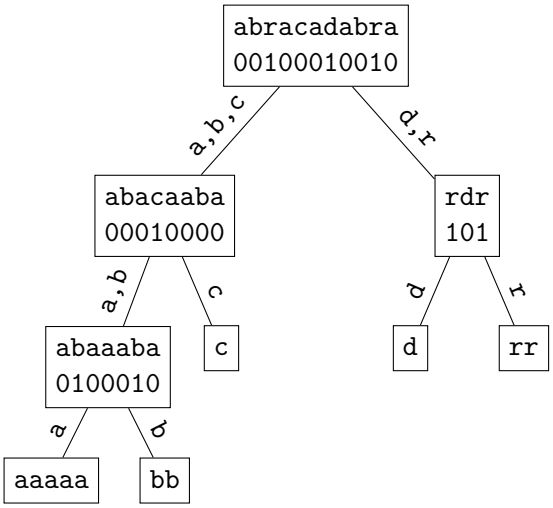

Figure 1.3: The wavelet tree for the string "abracadabra".

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 $i - \text{RANK}_1(i)$. If it is 1, the element is in the right child and the new index there is $i - \text{RANK}_0(i)$.

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 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 $\mathcal{O}(\log \sigma)$. ∎

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** $[l', r']$

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). ∎

Another commonly used method on wavelet trees is EXPAND given as pseudocode in Algorithm 8. If we are given an interval $[i, j]$ in the bitvector of some node $v = [l, r]$, EXPAND splits this interval into two intervals $[i_l, j_l]$ and $[i_r, j_r]$ of the left and right child.

Algorithm 8 Expands a range $[i, j]$ of node v into two child ranges.

EXPAND($v = [l, r], [i, j]$)
1 $i_l = \text{RANK}_0(i)$
2 $j_l = \text{RANK}_0(j)$
3 $i_r = \text{RANK}_1(i)$
4 $j_r = \text{RANK}_1(j)$
5 **return** $\langle [i_l, r_l], [i_r, j_r] \rangle$

Example 1.20
Let A be an integer array of size n. We want to answer range median queries with an $\mathcal{O}(n \log n)$ datastructure in time $\mathcal{O}(\log n)$. Here the median of a range $A[i, j]$ is defined as the $\lceil \frac{j-i+1}{2} \rceil$-th smallest element in $A[i, j]$.

The solution is given in Algorithm 9. It uses an integer wavelet tree and EXPAND to find the median.

Algorithm 9 Range median queries in an integer array.

```
RANGE-MEDIAN-QUERY([i, j])
 1  median = (j−i+1)/2
 2  v = [0, n − 1]                          // Root of the wavelet tree.
 3  range = [i, j]
 4  while not IS-LEAF(v)
 5      childRanges = EXPAND(v, range)
 6      if SIZE(childRanges[0]) > median
 7          v = LEFT-CHILD(v)
 8          range = childRanges[0]
 9      else
10          v = RIGHT-CHILD(v)
11          range = childRanges[1]
12  return SYMBOL(v)
```

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

Definition 1.21
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.

$$\mathcal{H}_0(S) = \sum_{\substack{c \in \Sigma \\ n_c > 0}} \frac{n_c}{n} \log \frac{n}{n_c} \tag{1.9}$$

The idea to compress S is to represent S as a wavelet tree using $\mathcal{H}_0$-compressed bitvectors. For a bitstring ω, S_ω we will denote the node in the wavelet tree containing all symbols from S whose binary representations starts with ω. So S_ε is the root, S_0 the left child of it and S_{10} the left child of the right child of the root.

Theorem 1.22
A sequence S of length n with alphabet σ can be represented with a wavelet tree in space $n\mathcal{H}_0(S)$ bits.

PROOF We will proof the claim by induction. Let H be the height of the wavelet tree. For a leaf S_c of the wavelet tree we get $\mathcal{H}_0(S_c) = 0$. Now as the induction hypothesis assume that the claim holds for all nodes S_ω where ω is a binary string of length L.

Now consider a subsequence S_ω corresponding to an inner vertex, where ω is a binary string of length $L - 1$. By our induction hypothesis the space needed to represent $S_{\omega 0}$ and $S_{\omega 1}$ is $n_{\omega 0}\mathcal{H}_0(S_{\omega 0})$ and $n_{\omega 1}\mathcal{H}_0(S_{\omega 1})$. Now the space needed to encode S_ω is (B_ω is

the bitvector in the node corresponding to S_ω):

$$\begin{aligned}\textsc{Space}(S_\omega) &= n_\omega \mathcal{H}_0(B_\omega) + n_{\omega 0}\mathcal{H}_0(S_{\omega 0}) + n_{\omega 1}\mathcal{H}_0(S_{\omega 1}) \\ &= n_{\omega 0}\log\frac{n_\omega}{n_{\omega 0}} + n_{\omega 1}\log\frac{n_\omega}{n_{\omega 1}} + n_{\omega 0}\mathcal{H}_0(S_{\omega 0}) + n_{\omega 1}\mathcal{H}_0(S_{\omega 1}) \\ &= \underbrace{n_{\omega 0}\log\frac{n_\omega}{n_{\omega 0}} + n_{\omega 0}\mathcal{H}_0(S_{\omega 0})}_{(a)} + \underbrace{n_{\omega 1}\log\frac{n_\omega}{n_{\omega 1}} + n_{\omega 1}\mathcal{H}_0(S_{\omega 1})}_{(b)} = (*)\end{aligned} \tag{1.10}$$

For (a) (and analogously (b)) we can now substitute the definition of $n_{\omega 0}\mathcal{H}_0(S_{\omega 0}) = \sum_{\alpha\in\sigma^{H-L}} n_{\omega 0\alpha}\log\frac{n_{\omega 0}}{n_{\omega 0\alpha}}$. In this sum, α are all possible ways to extend $\omega 0$ to a prefix corresponding to a vertex in the wavelet tree. We get:

$$\begin{aligned}(a) &= n_{\omega 0}\log\frac{n_\omega}{n_{\omega 0}} + \sum_{\alpha\in\sigma^{H-L}} n_{\omega 0\alpha}\log\frac{n_{\omega 0}}{n_{\omega 0\alpha}} \\ &= \sum_{\alpha\in\sigma^{H-L}} n_{\omega 0\alpha}\log\frac{n_\omega}{n_{\omega 0}} + \sum_{\alpha\in\sigma^{H-L}} n_{\omega 0\alpha}\log\frac{n_{\omega 0}}{n_{\omega 0\alpha}} \\ &= \sum_{\alpha\in\sigma^{H-L}} n_{\omega 0\alpha}\left(\log\frac{n_\omega}{n_{\omega 0}} + \log\frac{n_\omega 0}{n_{\omega 0\alpha}}\right) \\ &= \sum_{\alpha\in\sigma^{H-L}} n_{\omega 0\alpha}\left(\log n_\omega - \log n_{\omega 0} + \log n_{\omega 0} - \log n_{\omega 0\alpha}\right) \\ &= \sum_{\alpha\in\sigma^{H-L}} n_{\omega 0\alpha}\log\frac{n_\omega}{n_{\omega 0\alpha}}\end{aligned} \tag{1.11}$$

Plugging the values of (a) and (b) we get

$$\begin{aligned}(*) &= \sum_{\alpha\in\sigma^{H-L}} n_{\omega 0\alpha}\log\frac{n_\omega}{n_{\omega 0\alpha}} + \sum_{\alpha\in\sigma^{H-L}} n_{\omega 1\alpha}\log\frac{n_\omega}{n_{\omega 1\alpha}} \\ &= \sum_{\alpha'\in\sigma^{H-(L-1)}} n_{\omega\alpha'}\log\frac{n_\omega}{n_{\omega\alpha'}} \\ &= n_\omega\mathcal{H}_0(S_\omega)\end{aligned} \tag{1.12}$$

for the space of S_ω. ∎

1.7 Range Minimum Query

Definition 1.23
Given an array $A[0, n-1]$ from a totally ordered set. A *range minimum query* $\mathrm{RMQ}(i, j)$ calculates the index k of the smallest element $A[k]$ in a range $[i, j]$.

A complete precomputation would be able to answer the queries in $\mathcal{O}(1)$ but would need $\mathcal{O}(n^2 \log n)$ bits space.

Theorem 1.24
Range minimum queries can be answered in $\mathcal{O}(1)$ *using an index of size* $\mathcal{O}(n \log^2 n)$ *bits.*

PROOF Precalculate an array $M[0, n-1][\lceil \log n \rceil]$ with $M[i][j] = \mathrm{RMQ}(i, i + 2^j - 1)$. Then (using that min is idempotent)

$$\mathrm{RMQ}(i, j) = \min\{M[i][r], M[j - 2^i + 1][r]\} \tag{1.13}$$

where $r = \max\{r \mid 2^r \leq j - i + 1\}$. Table M is known as a *sparse table*. Range maximum queries work analogously. ■

Theorem 1.25
Range minimum queries can be answered in $\mathcal{O}(1)$ *per query using an index of size* $2n + o(n)$ *bits.*

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 known 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 $SA[sp_m], \ldots, SA[ep_m]$. Backward-Search is given in pseudocode in Algorithm 10. ∎

Algorithm 10 Backward search for a pattern P of length m.

BACKWARD-SEARCH(P)

```
1  sp = 0
2  ep = n − 1
3  for i = m − 1 downto 0
4        sp = C[P[i]] + RANK_c(sp, BWT)
5        ep = C[P[i]] + RANK_c(ep + 1, BWT) − 1
6  return (sp, ep)
```

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 are 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.

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$.

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.

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 11 allows to access arbitrary elements of the suffix array.

Algorithm 11 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
```

Theorem 2.23
If SA-order-sampling is used, the ACCESS-SAMPLED-SUFFIX-ARRAY*-method in Algorithm 11 takes at most $\mathcal{O}(s \cdot t_{LF})$ time, where t_{LF} is the time for an access in the LF array.*

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 11 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 12.

Algorithm 12 Access the sampled inverse suffix array.

```
Access-Sampled-Inverse-Suffix-Array(i)
1  idx = ⌈i/s⌉
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).

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.

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 $s = 2$.

2.5.1 Elias-Fano-Encoded CSA_Ψ

We can use Elias-Fano-Encoding to encode each of the increasing subsequences of Ψ, each in $[0, n-1]$. Let's calculate the needed space for each of these sequences. For each $c \in \Sigma$, n_c is the number of occurrences of c in the text (so the length of the increasing run corresponding to the suffixes starting with c).

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

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.

We can use Algorithm 13 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 $\mathcal{O}(m \log n)$.

Algorithm 13 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].
```

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 $\lfloor \log(x) \rfloor + 2\lfloor \log(\lfloor \log(x) \rfloor + 1) \rfloor + 1$ 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. For each character $c \in \Sigma$ we gap-encode its increasing Ψ sequence. 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. The space

for the compressed suffix array is then:

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

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. Then we split the summation into two sums and plug the definition for the zeroth-order entropy.

3 Range Queries

In this chapter different methods to count and report (weighted) points from a two-dimensional grid will be presented. Many string matching problems can be reduced to problems of this kind.

3.1 k^2-Trees

Definition 3.1
The *k^2-tree* is a datastructure that partitions an $s \times s$-grid into k^2 equal sized quadrants. This partition is represented as the root of a k^2-ary cardinal tree. The decomposition is done recursively on each subgrid until it has size 1×1.

For $k = 2$ the k^2-tree is also known as a *quadtree*. In the following examples we will always choose $k = 2$ for simplicity.

We will use k^2-trees to answer (weighted) two-dimensional range queries on a grid. Given a rectangular region $[x_1, x_2] \times [y_1, y_2]$, count/report the (top-$l$) points in it. A k^2-tree cannot count the points without traversing them all, so we will only consider report queries.

Although we cannot prove a good runtime for the k^2-tree or -treap, both perform quite well in practice.

3.1.1 Unweighted Points

To report all points in the given range $[x_1, x_2] \times [y_1, y_2]$ we traverse the k^2-tree. Starting with the root, at each node we check whether its child ranges (at least partially) overlap with the query range. If so, we step down into these children until ending at a leaf reporting its value. If a child does not overlap the query range, we do not step down into it. This is again shown in Algorithm 14.

3.1.2 Weighted Points

To handle weighted points, we annotate each node of the k^2-tree with the position and weight of the heaviest element in its corresponding subgrid (ties are broken arbitrarily).

Algorithm 14 Reports the points from k^2-tree T in range $[x_1, x_2] \times [y_1, y_2]$.

```
REPORT([x1, x2] × [y1, y2])
1  REPORT-RECURSIVE(K.root, [x1, x2] × [y1, y2])

REPORT-RECURSIVE(v, [x1, x2] × [y1, y2])
1  if s == 1
2      output v
3  else
4      for each child w of v
5          if w.range intersects [x1, x2] × [y1, y2]
6              REPORT-RECURSIVE(w, [x1, x2] × [y1, y2])
```

Then this element is removed from the grid, so that the children contain the next smaller maximal values of their corresponding subgrid. This gives the tree an additional max-heap structure, it is therefore also known as a *k^2-treap*.

Example 3.2
Consider the 8×8-grid shown in Figure 3.1. The corresponding 2^2-treap is shown in Figure 3.2. The root is annotated with $(3, 0) : 8$, meaning that the maximal value is 8 and is in the cell at coordinates $(3, 0)$ of the grid. Because it is then deleted, the maximum in the subgrid $[3, 4] \times [0, 1]$ is 2 (the only remaining value in this subgrid).

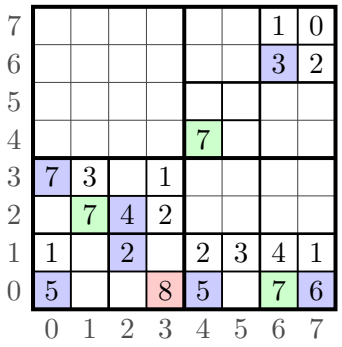

Figure 3.1: An 8×8 grid with weights in some cells. The thick lines show the subdivision for a 2^2-treap. The red cell is the overall maximum. After deleting it, the green cells are maximal in their corresponding 4×4 grid. After deleting them as well, the blue cells are maximal in their corresponding 2×2 grids.

To extract the heaviest points we use a max-priority-queue PQ. Initially we insert the root of the k^2-treap. Whenever we pop a vertex from PQ, we add all its non-empty children that intersect the query range into it. If the maximal element stored at the current vertex is inside the range, we add it to our result. Otherwise we ignore it and continue with the next node in the priority-queue. The pseudocode is given in Algorithm 15.

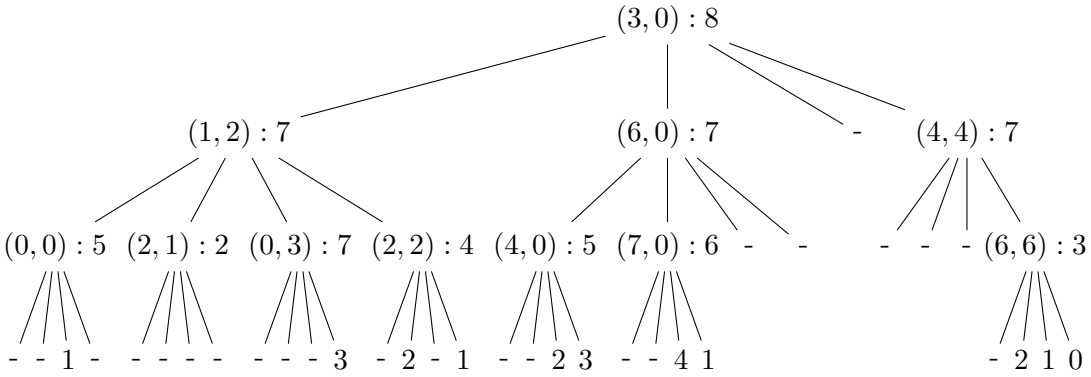

Figure 3.2: The 2^2-treap for the grid shown in Figure 3.1. The notation of the vertices shows first the position of the maximal element in the corresponding subgrid and then its value. The position is omitted on the last layer, since the corresponding subgrid has size 1×1.

3.2 Wavelet Trees

Different to k^2-trees we can prove good runtime for range queries when we use wavelet trees.

3.2.1 Unweighted Points

Definition 3.3
Given a $[0, n-1] \times [0, n-1]$ grid G and a set P of n points $(i, S[i])$ for $0 \leq i < n$. For a range of points $[x_1, x_2] \times [y_1, y_2]$ we define two queries:

- A *range count query* asks to count the number of points of G in the range.
- A *range report query* asks to report all of these points.

In above grids each row and column has only one point in it. This can be assumed without loss of generality by coordinate compression.

Theorem 3.4
Range count queries can be answered in $\mathcal{O}(\log n)$ time using an index of size $n \log n + o(n \log n)$ bits. Range report queries can be answered in $\mathcal{O}(\log n + occ \cdot \log n)$ using the same index. Here occ is the number of points in the given range.

Proof The grid has exactly one point per column, so we can store the points as a sequence of the y-coordinates. The x-coordinate is the position in this sequence. Now build an integer wavelet tree WT over this sequence. Each wavelet tree node corresponds to some range of y-coordinates. For example the left child of the root to $[y_1, y_2] = [0, \lfloor \frac{n}{2} \rfloor]$

Algorithm 15 Reports the top-l points from k^2-treap T in $[x_1, x_2] \times [y_1, y_2]$.

```
REPORT(x1, x2, y1, y2, l)
 1  PQ = empty max-priority-queue
 2  r = T.root
 3  PQ.insert(r)        // Priority-queue ordered by weight annotated to each vertex.
 4  c = 0
 5  while not PQ.empty() and c < l
 6      v = PQ.top()
 7      PQ.pop()
 8      if x1 ≤ v.x ≤ x2 and y1 ≤ v.y ≤ y2
 9          output v
10          c = c + 1
11      for each child w of v
12          if w.range intersects [x1, x2] × [y1, y2]
13              PQ.push(w)
```

and the right child of the root to $[y_1, y_2] = [\lceil \frac{n}{2} \rceil, n-1]$. The y-range covered by a wavelet tree node v is given by Y-RANGE(v). The x-range $[x_1, x_2]$ is just a range in the bitvector of the root node. By using EXPAND($v, [x_1, x_2]$), we can map this range to the two child values.

Method RANGE-COUNT from Algorithm 16 counts the number of occurrences by traversing the wavelet tree. When the y-range of a vertex v is complete inside the search range, all points in it belong to the search range. If it lies complete outside we can stop at this node. In the remaining case that the y-range is partially covering the search range, we step down into the two children.

Algorithm 16 Counts the points in range $[x_1, x_2] \times [y_1, y_2]$ using wavelet tree WT.

```
RANGE-COUNT([x1, x2] × [y1, y2])
1  return RANGE-COUNT(WT.root, [x1, x2] × [y1, y2])

RANGE-COUNT(v, [x1, x2] × [y1, y2])
 1  if x2 < x1
 2      return 0
 3  if [y1, y2] ∩ Y-RANGE(v) = ∅
 4      return 0
 5  if Y-RANGE(v) ⊆ [y1, y2]
 6      return x2 − x1 + 1
 7  l = LEFT-CHILD(v)
 8  r = RIGHT-CHILD(v)
 9  ⟨[x1^l, x2^l], [x1^r, x2^r]⟩ = EXPAND(v, [x1, x2])
10  return RANGE-COUNT(l, [x1^l, x2^l] × [y1, y2]) + RANGE-COUNT(r, [x1^r, x2^r] × [y1, y2])
```

RANGE-COUNT needs time in $\mathcal{O}(\log n)$, which is the maximal number of vertices needed to span the y-range of the query.

To report all the points in the search range, we need to change Line 6 in Algorithm 16 to call RANGE-REPORT$(v, [x_1, x_2])$. The code is given in Algorithm 17.

Algorithm 17 Reports the points in $[x_1, x_2]$ of a node with y-range completely covered.

```
RANGE-REPORT(v, [x_1, x_2])
 1  if x_2 < x_1
 2      return 0
 3  if IS-LEAF(v)
 4      y = Y-RANGE(v)[0]
 5      for x = x_1 to x_2
 6          output (SELECT_y(x + 1), y)
 7      return x_2 − x_1 + 1
 8  else
 9      l = LEFT-CHILD(v)
10      r = RIGHT-CHILD(v)
11      ⟨[x_1^l, x_2^l], [x_1^r, x_2^r]⟩ = EXPAND(v, [x_1, x_2])
12      return RANGE-REPORT(l, [x_1^l, x_2^l]) + RANGE-REPORT(r, [x_1^r, x_2^r])
```

RANGE-REPORT from Algorithm 17 together with RANGE-COUNT runs in $\mathcal{O}(\log n + occ \cdot \log n)$, because for each of the occ occurrences we must step down to a leaf of the wavelet tree in time $\mathcal{O}(\log n)$. ∎

Example 3.5
Position Restricted Substring Search: Let T be a text of length n and q be a query of length m.

- A count query asks for the number of occurrences of q in $T[l, r]$.
- A report query asks for a list of all occurrences of q in $T[l, r]$.

In both cases the solution is to build a wavelet tree WT over the suffix array SA. Now we use BACKWARD SEARCH to get the suffix array interval $[x_1, x_2]$ of q and make a count/report query on WT with range $[x_1, x_2] \times [l, r]$. The time complexity is $\mathcal{O}(m \cdot t_{LF} + \log n + occ \log n)$ where the first summand is the time needed to get the suffix array interval. The space needed is just the space for the wavelet tree and for the suffix array, so $n \log n + o(n \log n) + |CSA|$ bits.

Example 3.6
Let T be a text of length n and q be a query of length m.

- A substring rank query RANK$_q(i, T, [l, r])$ asks for the number of occurrences of q in $T[l, r]$.
- A substring select query SELECT$_q(i, T, [l, r])$ asks for the i-th occurrence of q in T.

We will use the following datastructures as our index: The wavelet tree from Theorem 2.18 to access SA, CSA, LF and Ψ in $\mathcal{O}(\log n)$ each. This needs $n \log n + o(n \log n)$ bits space. Further we need the C array we used with the suffix array. For each $c \in \Sigma$ it stores the position of the first suffix in SA starting with c.

To answer the queries, we first get the suffix array interval $[x_1, x_2]$ of q. This can be done with either backward or forward search. The interval gives us all suffixes starting with q. Now for $\textsc{Select}_q$ we want are interested in the i-th smallest index in this interval. This can be done equivalently to the calculation of range median queries in Example 1.20. For $\textsc{Rank}_q$ we want to count how many smaller elements there are. When in the leaf of the wavelet tree, we can go back up to the root, always adding the number of smaller elements from nodes further left in the wavelet tree.

4 Document Retrieval

We are given a collection $D' = \{d_1, \ldots, d_{N-1}\}$ of documents. Each of d_i is a string over an alphabet $\Sigma' = [2, \sigma]$ terminated by a sentinel symbol 1 (or #). We define $D = D' \cup d_0$ with a sentinel document $d_0 = 0$. We now want to answer word queries $Q = \{q_0, \ldots, q_{m-1}\}$. Q is called *bag of words* and is an unordered set of size m.

Definition 4.1
Given a collection D, a query Q of length m and a similarity measure $\mathcal{S} : D \times \mathcal{P}_{=m}(\Sigma') \to \mathbb{R}$. Calculate the *top-k documents* of D with regard to Q an $\mathcal{S}$.
That is a sorted list $T = \{\tau_0, \ldots, \tau_{k-1}\}$ with $\mathcal{S}(d_{\tau_i}, Q) \geq \mathcal{S}(d_{\tau_{i+1}}, Q)$ for $0 \leq i < k$ and $\mathcal{S}(d_{\tau_{k-1}}, Q) \geq \mathcal{S}(d_j, Q)$ for $j \notin T$.

4.1 Similarity Measures

Definition 4.2
The following document dependent factors appear in many different similarity measures $\mathcal{S}$:

- $f_{d,q}$ is the *term frequency*. It counts the number of times, word q appears in document d.
- $F_{\mathcal{D},q}$ is the *document frequency*. It counts the number of distinct documents from $\mathcal{D}$ containing q at least once.

Definition 4.3
The *Okapi BM25* similarity measure is given by:

$$\mathcal{S}^{BM25}_{Q,d} = \sum_{q \in Q} \frac{(k_1 + 1)\, f_{d,q}}{k_1 \left(1 - b + \frac{n_d}{n_{avg}}\right) + f_{d,q}} \cdot f_{Q,q} \cdot \ln \frac{N - F_{\mathcal{D},q} + 0.5}{F_{\mathcal{D},q} + 0.5} \tag{4.1}$$

Here n_d is the length of document d and n_{avg} is the average document length.

There are several other possible ideas to build a similarity measure:

- Documents can be assigned *static weights*. An example is the Page-Rank algorithm.
- A *language model* can be used to compute the probability to generate the query using the text statistics of each document.

- In a *vector space model* we compute the cosine of the angle in σ-dimensional space between a query vector and a document vector.
- *Zone ranking* weighs words appearing in the title of a web page weigh more than words in the body.

4.2 Inverted Index

Definition 4.4
In an *inverted index* (IVI) for each term q (excluding the sentinels) a list of pairs of document id and term frequency is stored. The pairs are ordered according to their document ids. Further for each term the document frequency is stored.

To process a query we sequentially iterate through the lists containing the query phrases and calculate the ranking function. The query complexity therefore depends on the document frequency.

It is not possible to answer phrase queries with this variant of an inverted index. Direct support of arbitrary phrase queries would require $\mathcal{O}(n^2)$ lists.

Example 4.5
Consider the three documents $\mathcal{D} = \{d_1, d_2, d_3\}$:

$$\begin{aligned} d_1 &: \text{is big data really big} \\ d_2 &: \text{is it big in science} \\ d_3 &: \text{big data is big} \end{aligned}$$

We will get the following inverted index:

$$\begin{aligned} \text{big} &: \{(1,2),(2,1),(3,2)\} & F_{\mathcal{D},\text{big}} &= 3 \\ \text{data} &: \{(1,1),(3,1)\} & F_{\mathcal{D},\text{data}} &= 2 \\ \text{in} &: \{(2,1)\} & F_{\mathcal{D},\text{in}} &= 1 \\ \text{is} &: \{(1,1),(2,1),(3,1)\} & F_{\mathcal{D},\text{is}} &= 3 \\ \text{really} &: \{(1,1)\} & F_{\mathcal{D},\text{really}} &= 1 \\ \text{science} &: \{(2,1)\} & F_{\mathcal{D},\text{science}} &= 1 \end{aligned}$$

Theorem 4.6
An inverted index can be represented in $n\mathcal{H}_0(\mathcal{D}) + 3n + o(n) + \mathcal{O}(\sigma \log n)$ *bits, where* $n = \sum_{d\in\mathcal{D}} n_d$ *and* $f_{\mathcal{D},q} = \sum_{d\in\mathcal{D}} f_{d,q}$.

Proof For each term we use Elias-Fano-Encoding to store the increasing list of document ids. The list of frequencies itself is unary encoded decreased by one (each frequency x

is encoded by x bits). With this representation we get:

$$\begin{aligned}\textsc{Space}(\text{IVI}) &\overset{2.26}{=} \sum_{q\in\Sigma} \underbrace{2F_{\mathcal{D},q} + F_{\mathcal{D},q}\log\frac{N}{F_{\mathcal{D},q}} + o(F_{\mathcal{D},q})}_{\text{document ids}} + \underbrace{f_{\mathcal{D},q}}_{\text{frequencies}} + \underbrace{\mathcal{O}(\log n)}_{\text{pointer}} \\ &\leq 3n + o(n) + \mathcal{O}(\sigma\log n) + \sum_{q\in\Sigma} F_{\mathcal{D},q}\log\frac{n}{F_{\mathcal{D},q}} \\ &\overset{(*)}{\leq} 3n + o(n) + \mathcal{O}(\sigma\log n) + n\sum_{q\in\Sigma}\frac{f_{\mathcal{D},q}}{n}\log\frac{n}{f_{\mathcal{D},q}} \\ &= n\mathcal{H}_0(\mathcal{D}) + 3n + o(n) + \mathcal{O}(\sigma\log n)\end{aligned} \tag{4.2}$$

In $(*)$ we assumed that $f_{\mathcal{D},q} < \frac{n}{2}$ for all q. When the inverted index contains a sufficient amount of documents, this is safe to assume. ■

4.3 GREEDY framework

Definition 4.7
Let $\mathcal{D}$ be the *document array* of length n. For each suffix $SA[i]$ the document array $\mathcal{D}[i]$ contains the identifier of the document, in which suffix $SA[i]$ starts. A suffix array (or suffix tree) with this information added is called *generalized suffix array* (or *generalized suffix tree*).

Definition 4.8
The *GREEDY framework* for single term $f_{d,q}$-ranking of Culpepper et al. [1] consists of:

- A compressed suffix array CSA of concatenation $\mathcal{D}$.
- A wavelet tree WTD of the document array of D.

The pseudocode Ranked-Search for a query q is given in Algorithm 18. The first step is a backward search in the compressed suffix array CSA to get the interval $[l, r]$ of all occurrences of q. This interval is stored in a priority queue. As long as this is not empty and we do not yet extracted k documents, we take the longest interval from the priority queue. If this is a leave, we output its corresponding symbol as a hit. Otherwise we split the interval into the corresponding intervals of the two children in the wavelet tree WTD.

Example 4.9
Consider the text $T = \omega_2\omega_1\omega_3\omega_3\#\omega_1\omega_1\omega_4\omega_1\#\omega_1\omega_4\omega_3\omega_1\#\omega_5\omega_5\#$ with $\# < \omega_1 < \ldots < \omega_5$. The suffix array is $SA = \{4, 9, 14, 17, 8, 13, 5, 6, 1, 10, 12, 0, 3, 2, 7, 11, 16, 15\}$ and the document array is $\mathcal{D} = \{0, 1, 2, 3, 1, 2, 1, 1, 0, 2, 2, 0, 0, 0, 1, 2, 3, 3\}$. The red parts are the range Backward-Search returned for query ω_1. The interval in $\mathcal{D}$ corresponds to the multiset of all documents containing ω_1. Figure 4.1 shows the wavelet tree over $\mathcal{D}$ with the interesting intervals marked in red. The leave intervals are extracted in order $1, 2, 0$. So for a top-2-query, we would have found d_1 (3 occurrences) and d_2 (2 occurrences).

Algorithm 18 Search for top-k documents containing q.

RANKED-SEARCH(q, k)

```
1  [l, r] = BACKWARD-SEARCH(CSA, q)
2  pq.push(⟨r − l + 1, [l, r], WTD.root()⟩)        // Max-PQ sorted by interval size.
3  h = 0
4  while h < k and not pq.empty()
5      ⟨s, [l, r], v⟩ = pq.pop()
6      if WTD.is_leaf()
7          output ⟨WTD.symbol(v)⟩
8          h = h + 1
9      else
10         ⟨⟨[l_l, r_l], v_l⟩⟨[l_r, r_r], v_r⟩⟩ = WTD.expand(v, [l, r])
11         pq.push(⟨r_l − l_l + 1, [l_l, r_l], v_l⟩)
12         pq.push(⟨r_r − l_r + 1, [l_r, r_r], v_r⟩)
```

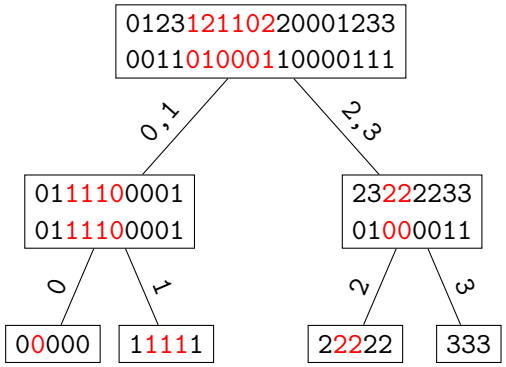

Figure 4.1: The wavelet tree over the document array $\mathcal{D}$. The interval with the documents corresponding to query $q = \omega_1$ are marked in red.

Theorem 4.10
The GREEDY framework needs time $\mathcal{O}(N \cdot (t_{deleteMax} + t_{insert}))$ in the worst case, where N is the number of documents in the collection and $t_{deleteMax}$ and t_{insert} are the times needed for the corresponding wavelet tree operations.

PROOF Consider a collection $\mathcal{D}$ where each $d \in \mathcal{D}$ is the same, for example $d = a$. Let the query be a as well.

The backward search will return the whole set of all documents (excluding the sentinels), because every document contains the query. Not the GREEDY framework steps through the wavelet tree until k leaves are found. In each step it takes the biggest interval from the priority queue and inserts its two children. Since the intervals in the leaves are of size 1 each, first all inner nodes must be processed. For N documents in the wavelet tree there are $N - 1$ inner nodes. Therefore the runtime is $\mathcal{O}(N \cdot (t_{deleteMax} + t_{insert}))$. ∎

4.4 Document Frequency $F_{\mathcal{D},q}$

In this section we will see how to efficiently compute the document frequency $F_{\mathcal{D},q}$ for a given query q following a method from Sadakane [9].

Definition 4.11
The *binary generalized suffix tree* ($BGST$) of a text is the suffix tree with inserted inner nodes, such that the each vertex has degree 0 or 2. Figure 4.2 shows the $BGST$ for `LA O LA # O LA LA LA # O O LA # $`.

To count the document frequency $F_{\mathcal{D},q}$, execute the following steps:

1. Build the $BGST$.
2. For each inner node v in $BGST$ keep a list L_v of repeated documents. A document d is added to L_v, if d occurs in a leaf of the left and of the right subtree.
3. For a pattern q let v_q be the *locus*, that is the lowest node which path is prefixed by q. $F_{\mathcal{D},q}$ equals the number of leaves in the subtree of v_q minus the number of repeated documents ($\sum_{v \in T_{v_q}} |L_v|$) in v_q's subtree T_{v_q}.

To find the number of repeated documents in the subtree of an inner node, number the nodes in order (as in Figure 4.2). Traverse in this order and append $|L_v|$ in unary coding ($|L_v|$ 0s and one 1) to a bitvector H, which was initialized with a single 1. Then all nodes of each possible subtree are contiguous and the number of repeated documents can be calculated by two select queries. This is shown in Algorithm 19. The runtime depends only on the time to do the backward search.

Algorithm 19 Compute the document frequency $F_{\mathcal{D},q}$ for query q.

```
DOCUMENT-FREQUENCY(q)
1  [l, r] = BACKWARD-SEARCH(CSA, Q)
2  s = r − l + 1
3  y = SELECT_1(r, H)
4  if l == 0
5       return s − (y − r + 1)
6  else
7       x = SELECT_1(l, H)
8       return s − (y − r + 1 − (x − l + 1))
```

Example 4.12
Figure 4.2 shows the $BGST$ for `LA O LA # O LA LA LA # O O LA # $`. We get the following bitvector H:

$$H = 1\underbrace{1}_{v_1}\underbrace{1}_{v_2}\underbrace{1}_{v_3}\underbrace{0001}_{v_4}\underbrace{1}_{v_5}\underbrace{1}_{v_6}\underbrace{01}_{v_7}\underbrace{01}_{v_8}\underbrace{01}_{v_9}\underbrace{0001}_{v_{10}}\underbrace{1}_{v_{11}}\underbrace{1}_{v_{12}}\underbrace{01}_{v_{13}}$$

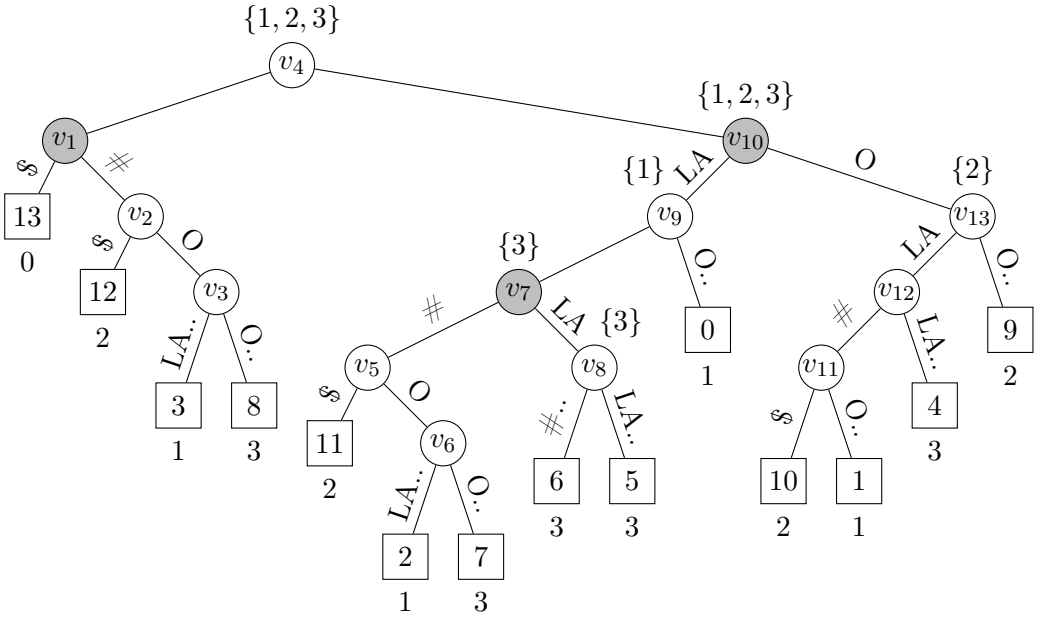

Figure 4.2: The binary generalized suffix tree of an example document collection $\mathcal{D} =$ `LA O LA # O LA LA LA # O O LA # $`. The filled vertices are the dummy nodes added to get a binary tree. Round nodes are inner nodes, square nodes are leaves corresponding to suffixes. The number below each leaf is the entry in the document array.

Let's now consider $q =$ `LA`. Then BACKWARD-SEARCH returns suffix array interval $[4, 9]$. Executing the steps in Algorithm 19 yields $s = 6$, $y = 15$ and $x = 7$, so that 3 is returned.

Theorem 4.13
The document frequency can be calculated with an index needing $\mathcal{O}(n + |CSA|)$ bits space.

PROOF The bitvector H has maximum size $2n - N$. Additionally $o(n)$ bits for a constant time select structure are needed. The only thing left is the space needed for CSA. ∎

4.4.1 Calculating $F_{\mathcal{D}_v,q}$

Let $F_{\mathcal{D}_v,q}$ be the subset of documents which are represented by a node in the wavelet tree over the document array. To compute it efficiently we introduce another array (following Gog [3]):

Definition 4.14
The *repetition array* R contains for each 0 in H the corresponding repeated element of the associated node.

We can build a wavelet tree WTD over the document array and a wavelet tree WTR over the repetition array. Using SELECT_1 we can as above convert an interval $[l, r]$ of the suffix

array into ranges in the two wavelet trees. When traversing WTD we can simultaneously traverse WTR. The size of the range in WTR always counts the number of repetitions.

Example 4.15
Continuing above example, we get for H and R:

$$H = 1\underbrace{1}_{v_1}\underbrace{1}_{v_2}\underbrace{1}_{v_3}\underbrace{0001}_{v_4}\underbrace{1}_{v_5}\underbrace{1}_{v_6}\underbrace{01}_{v_7}\underbrace{01}_{v_8}\underbrace{01}_{v_9}\underbrace{0001}_{v_{10}}\underbrace{1}_{v_{11}}\underbrace{1}_{v_{12}}\underbrace{01}_{v_{13}}$$

$$R = \underbrace{123}_{v_4}\underbrace{3}_{v_2}\underbrace{3}_{v_8}\underbrace{1}_{v_9}\underbrace{123}_{v_{10}}\underbrace{2}_{v_{13}}$$

4.5 Document Listing

Definition 4.16
In *document listing* we ask for all the distinct documents containing a query q.

We will present a solution by Muthukrishnan [7] which has an optimal running time in the number of different documents. The pseudocode is given in Algorithm 20.

- Precompute a text index (i.e. CSA) and document array $\mathcal{D}$. Further precompute an array E with $E[i] = \max\{j \mid j < i \wedge D[j] = D[i]\}$.
- For a query q get the lexicographical range of $[l, r]$ of q. Use range minimum queries on E to get the distinct documents in the lexicographical range of q. When the RMQ result gets bigger than l, we can stop, because this document was found before.

Algorithm 20 List all documents containing q.

```
DOCUMENT-LISTING(q)
1  [i, j] = BACKWARD-SEARCH(CSA, q)
2  DOCUMENT-LISTING-RECURSIVE([i, j], i)

DOCUMENT-LISTING-RECURSIVE([i, j], sp)
1  if j ≥ i
2      p = RMQ(i, j)
3      if E[p] < sp
4          output D[p]
5      DOCUMENT-LISTING-RECURSIVE([i, p − 1], sp)
6      DOCUMENT-LISTING-RECURSIVE([p + 1, j], sp)
```

Example 4.17
Consider the concatenation

$$\mathcal{C} = \underbrace{\texttt{ATA\#}}_{d_1}\underbrace{\texttt{TAAA\#}}_{d_2}\underbrace{\texttt{TATA\#}}_{d_3}\texttt{\$}$$

of length $n = 15$. See Table 4.1 for the needed values. BACKWARD-SEARCH returns the gray shaded interval for the query $q =$ `TA`. The minimal value in E is at $i = 13$, so the document d_2 at this position contains q. The next bigger value is at $i = 11$, so q is also in document d_3. The next bigger value is at $i = 12$ adding document d_1 to the result. The next bigger value is at $i = 14$, with $E[14] = 11$ which is greater than or equal to the lower bound of the interval found in the backward search, so we can stop here.

i	=	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SA	=	14	13	3	8	12	2	7	6	5	10	0	11	1	4	9
D	=	0	3	1	2	3	1	2	2	2	3	1	3	1	2	3
E	=	−1	−1	−1	−1	1	2	3	6	7	4	5	9	10	8	11

Table 4.1: Suffix array, document array and E for string $C =$ `ATA#TAAA#TATA#$`.

4.6 Top-k Single Term Frequency

Definition 4.18
Given a query term (or phrase) q of length m and a parameter k. Report the *top-k documents with respect to single term frequency*.

Theorem 4.19
The top-k documents can be found in time $\mathcal{O}(m+x(\log k+\log N))$, where x is the number of distinct documents q appears in.

PROOF The documents can be found in the following steps:

1. Get the x distinct documents $d_{r_1}, \ldots, d_{r_x}$ where q occurs in using the method from Section 4.5 in $\mathcal{O}(m)$.
2. Determine the frequency $f_{d_{r_i},q}$ of q in each d_{r_i}. This can be done in $\mathcal{O}(\log N)$ per document.
3. Maintain a min-heap of $(f_{d_{r_i},q}, d_{r_i})$-pairs of size k. The heap operations take time $\mathcal{O}(\log k)$.

In total we get $\mathcal{O}(m + x \cdot \log N + x \cdot \log k) = \mathcal{O}(m + x(\log k + \log N))$. ∎

The solution presented in Theorem 4.19 is already independent of the text length n. However it is still dependent on x, which can in the worst case be as large as the number of documents N.

Theorem 4.20
The top-k documents can be found in time $\mathcal{O}(m \cdot t_{LF} + \log n \cdot k)$ (Gog [2]).

Proof We will prove this theorem with a running example: Consider the concatenation $C =$ `ATA#TAAA#TATA#$` of length $n = 15$ consisting of three documents. The first step is to build the generalized suffix tree GST of C. It is shown in Figure 4.3. A document id i is added to an inner node v, if v is the lowest common ancestor of two leaves marked by i. These document ids are shown in red.

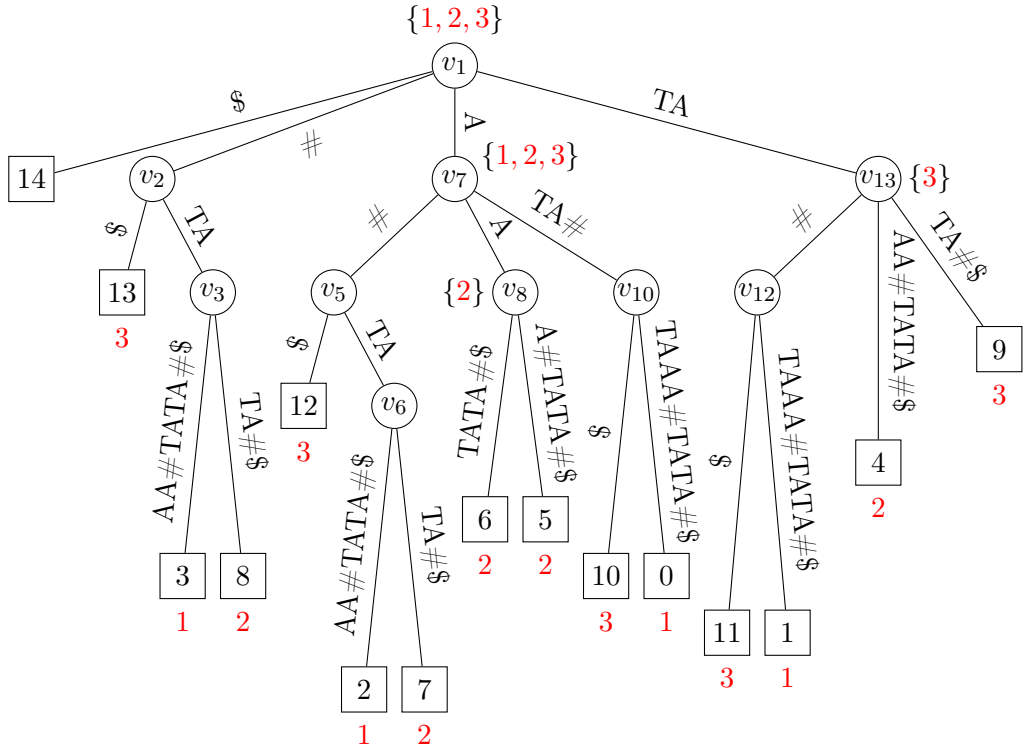

Figure 4.3: The generalized suffix tree for concatenation $C =$ `ATA#TAAA#TATA#$`. The red numbers on the leaves are the document array. The red marks on the inner vertices show the lowest common ancestor of two equal elements in the document array (as in document listing in Section 4.5).

In the next step the nodes need to be numbered. For each inner node v assign it the identifier $id(v)$ which is the index of the rightmost leaf in the leftmost subtree of v plus 1. This numbering has the following properties:

1. $id(v) \neq id(w)$ for all $v \neq w$. To show that this is true, we need to distinguish two cases: Assume $\text{LCA}(v, w) \notin \{v, w\}$. Then they have disjoint subtrees, so they get a different id. If however $\text{LCA}(v, w) \in \{v, w\}$, then w.l.o.g. v is in w's subtree. But if they got the same id, then v could have at most one child. If it had none, it is a leaf and would not be numbered. If it had exactly one, it would not even be part of the suffix tree.

2. $id(v) \in [1, n]$ for all v. However, not all numbers from $[1, n]$ must appear as ids.

3. $id(v) \in [lb(v), rb(v)]$, where $lb(v)$ and $rb(v)$ are the indices of the leftmost and rightmost leaves in v's subtree ($[lb(v), rb(v)]$ is v's suffix array interval).

Next we connect each document id i at some node v_x to the closest ancestor $_y$ that also contains document id i as shown in Figure 4.4. This allows the following observations:

- The subset of nodes marked with document id i correspond to the vertices of the suffix tree of document i.
- Document id i occurs at most n_i times in the leaves of the generalized suffix tree and therefore at most $n_i - 1$ times as an inner node. Summing up, this gives a total of at most $2n - N$ document labels in the whole GST.

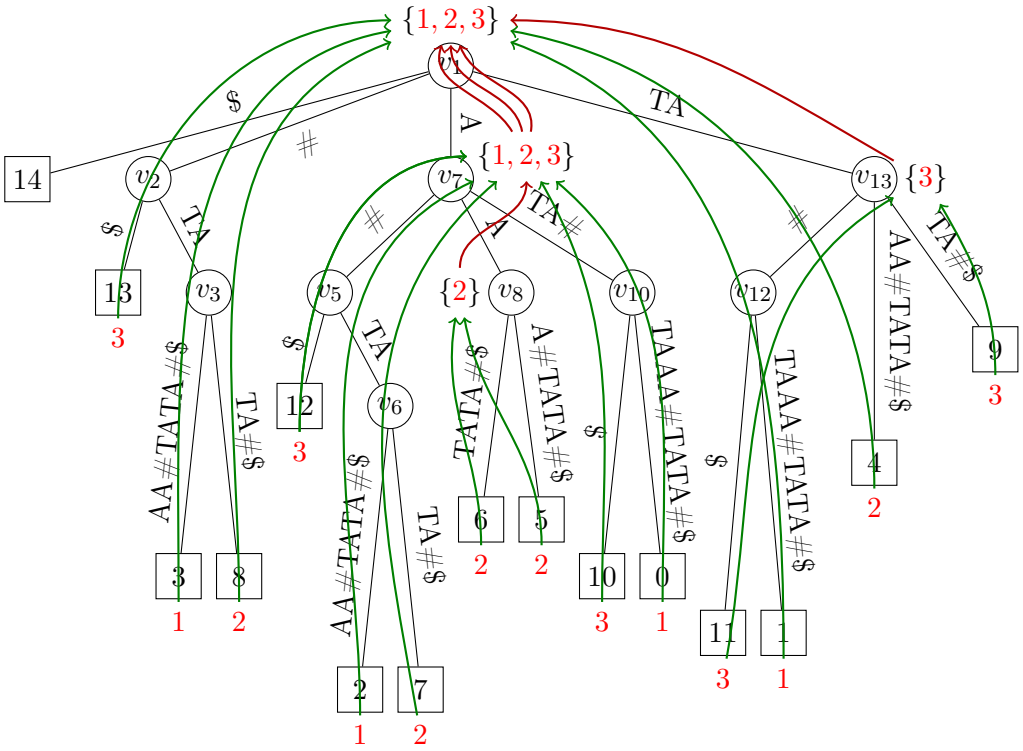

Figure 4.4: The suffix tree from Figure 4.3 with added arrows between document ids. The red arrows have weight at least 2.

We now have the index to answer queries. Consider a query q. We need to find the locus for q. Again this is the first node v, such that the path to v is prefixed by q. Then we observe:

- Per document i there is at most one arrow leaving v's subtree upwards. For each of these pointers we associate a weight: For a pointer of document id i, the weight is equal to the number of times document i appears in the subtree of v.

- The pointer of document i leaving the locus v has the biggest weight among all pointers for document i in v's subtree.
- Enumerating all the pointers leaving v's subtree is equivalent to document listing. Top-k frequency corresponds to retrieving the k heaviest pointers (and respectively their corresponding documents).

We will now describe how the problem to find the heaviest pointers can be mapped to an orthogonal range queries problem. We already know of efficient solutions for that.

From now on we will only consider arrows with a weight at least 2. In Figure 4.4 this is the red subset of the arrows. All other (green) arrows correspond to documents where a query appears only once. If our method considering only the heavy arrows returns less than k documents, we can use the classical document listing structure to find more documents.

Each pointer is assigned a (x, y)-coordinate. For this we need a bitvector H similar to the one used for document listing. It is generated by first writing n 1s and then inserting a 0 right before the i-th 1 per upward arrow leaving node v_i. Obviously the 0s in H correspond to the pointers and the pointer corresponding to the i-th 0 gets x-coordinate i. The y-coordinate is the depth of the target node of the pointer. In our example we get the values in Table 4.2. The points are plotted into a grid in Figure 4.5.

H	=	1	1	1	1	1	1	0	0	0	1	0	1	1	1	1	1	0	1	1	1
$\mathcal{D}$	=							1	2	3		2						3			
x	=							0	1	2		3						4			
y	=							0	0	0		1						0			
w	=							2	3	2		2						2			

Table 4.2: The mapping of pointers to coordinates for the example suffix tree in Figure 4.4.

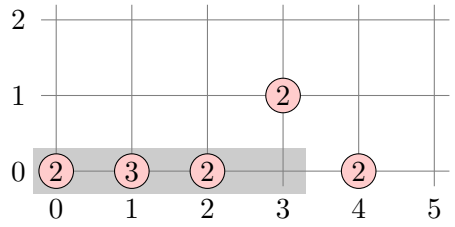

Figure 4.5: The points corresponding to the pointers with weight at least two plotted into a 2-dimensional grid. The shaded area corresponds to the query range $[0, 3] \times [0, 0]$

Assume the query $q =$ `A` and Backward Search returns a suffix array interval $[l, r] = [4, 10]$ for it. These are the positions of the leftmost and rightmost leaves descending

form locus $v = v_7$ (note that v is not computed and not known). Via two select queries interval $[l, r]$ can be mapped to an interval $[l', r'] = [0, 3]$ in the x-coordinates. We extract the positions of the zeros in H in $[l, r]$ as shown in Algorithm 21. For the y-coordinates the lower bound is always 0. For the upper bound we remember that the y-coordinate of the plotted points is the depth of the target node and that we only want to find arrows ending above the locus. Therefore we can take the length $|q| - 1$ as the upper y-coordinate. We reduced the query operation to a 3-sided range query for the k heaviest points in $[l', r'] \times [0, |q| - 1]$. This can be solved with the known range query methods from Section 3. For our example the query range is shown shaded in Figure 4.5.

Algorithm 21 Maps a suffix array interval $[l, r]$ to an x-interval $[l', r']$.

MAP-INTERVAL($[l, r]$)

```
1  [l', r'] = [0, -1]
2  if l > 0
3        l' = SELECT_1(l - 1, H) - (l - 1)
4  if r > 0
5        r' = SELECT_1(r - 1, H) - r
6  return [l', r']
```

■

At last, an optimal solution is known but is not covered in this document.

Theorem 4.21
The top-k documents can be found in time $\mathcal{O}(m + k)$ query time in $\mathcal{O}(n \log n)$ bits space (Navarro & Nekrich [8] based on framework by Hon [4]).

5 External Memory

When handling very large texts and indices, the data might not fit into memory any more. In this case some of it must be swapped to the disk, resulting in much greater access times. To analyze the performance of these indices we introduce the *external memory model* shown in Figure 5.1. Here the CPU can only work with data available in the main memory but all executed operations have zero cost. The only costing operation is to load a block from the disk to the main memory. Each of these loads costs one unit. In the following sections we specify the model by two parameters: B is the number of items stored in each block and M is the number of items that fit into the main memory simultaneously.

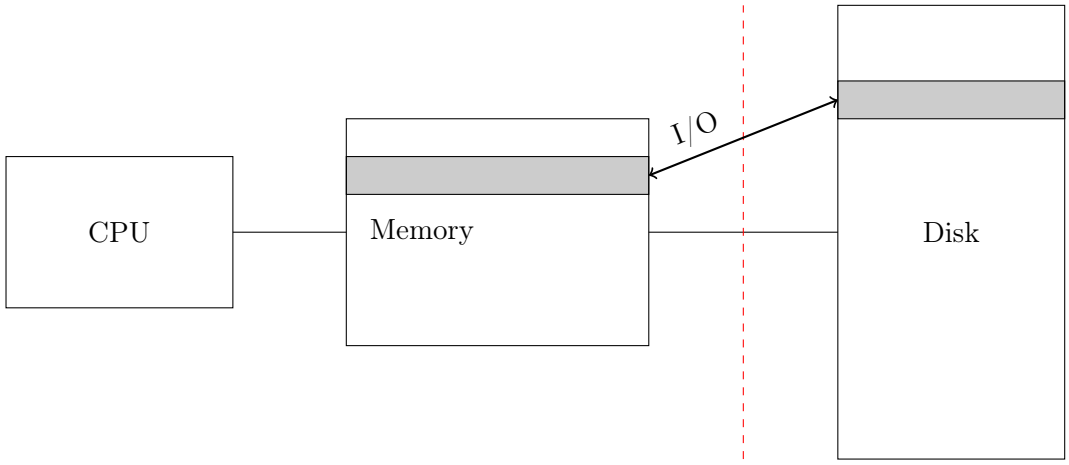

Figure 5.1: Schematic set up of the external memory model.

Example 5.1
Let T be a text of length n and q be a query of size m. The I/O complexity for count queries using a suffix array is given by $\mathcal{O}(m \cdot \log \frac{n}{B})$. For this example we used a naive binary search in the suffix array, comparing the query to $\mathcal{O}(\log n)$ suffixes.

5.1 String-B-Tree

Given is a set $S = \{s_0, \ldots, s_{N-1}\}$ of N strings of total length N. A *prefix count query* asks for the subset of strings starting with a given pattern P. We will describe a datastructure to answer prefix count queries efficiently in the external memory model.

Definition 5.2
A *String-B-Tree* (*SBT*) is a combination of a B-tree and a Patricia-trie. The keys of the *SBT* are pointers to the text (which is stored in external memory). The keys are stored in the leaves and their order is determined by the lexicographical ordering of the corresponding strings. The inner vertices store the minimal and the maximal keys of their children to allow efficient navigation.

Each node v of the *SBT* is stored in a disk block and contains an ordered string set $S_v \subset S$. For a $b \in \Theta(B)$ we assume $b \leq |S_v| \leq 2b$. The leftmost string in S_v is $L(v)$ and the rightmost one is $R(v)$.

To construct the *SBT* partition the ordered S into groups of b strings (the last group may have less than b strings). Each group is mapped to a leaf, such that a left-to-right scan of the leaves gives all strings from S in lexicographical order. With each pair S_j, S_{j+1} we associate the longest common prefix $\mathrm{LCP}(S_j, S_{j+1})$. Each internal node v had $d(v)$ children $u_0, \ldots, u_{d(v)-1}$ where $\frac{b}{2} \leq d(v) \leq b$ (the root may have from 2 to b children). Its set S_v is formed by copying the leftmost and rightmost strings of v's children: $S_v = \{L(u_0), R(u_0), \ldots, L(u_{d(v)-1}), R(u_{d(v)-1})\}$

Example 5.3
Consider the following text where the numbers above each word give the index of its first character. Its *STB* is given in Figure 5.2.

```
                1   1     2     2     3     4
0 2     8       3   7     3     9     6     2
i think that the ruddy widow really wants ripe

4           5   6   6     7     7     8
7           8   2   6     2     7     4
watermelon and red roses when winter arrives
```

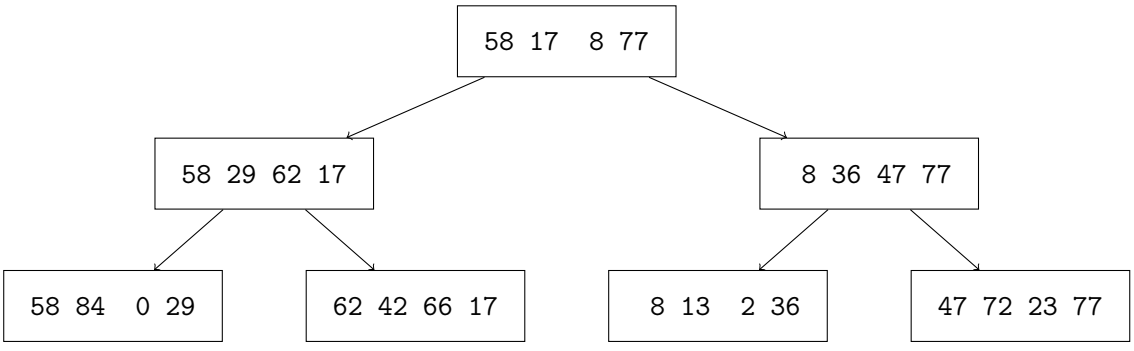

Figure 5.2: The *STB* for text T = `i think that the ruddy widow really wants ripe and red roses when winter arrives.`

In each node v of the SBT its string set S_v is stored as a *blind trie*. To do this we build a patricia trie over the (binary encoded) strings in S_v. We can store the trie in a succinct tree representation, so that it fits into the same memory block and the matching in it does not require further I/Os. For a pattern P we follow the blind search until reaching a leaf. We then check in the text, whether P occurs at the pointer returned by the blind search in the leaf and either stop or proceed with the left or right child.

Assuming that the subtree sizes are stored at each node, count queries for a pattern P then take $\mathcal{O}(\frac{|P|}{B} + \log_B N)$ time. For locate queries we need to descend to each leaf containing P, so the time becomes $\mathcal{O}(\frac{|P|+Z}{B} + \log_B N)$ where Z is the number of strings in S prefixed by P.

The SBT has $\mathcal{O}(\frac{N}{B})$ leaves and therefore $\mathcal{O}(\frac{N}{B})$ inner nodes. Each node fits into a disk block.

5.2 Geometric Burrows-Wheeler-Transformation

Index

Bibliography

[1] J. Shane Culpepper et al. "Top-k Ranked Document Search in General Text Databases". In: *Algorithms – ESA 2010*. Ed. by Mark de Berg and Ulrich Meyer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 194–205. ISBN: 978-3-642-15781-3.

[2] Simon Gog and Gonzalo Navarro. "Improved Single-Term Top-<italic>k</italic> Document Retrieval". In: *2015 Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pp. 24–32. DOI: 10.1137/1.9781611973754.3. eprint: http://epubs.siam.org/doi/pdf/10.1137/1.9781611973754.3. URL: http://epubs.siam.org/doi/abs/10.1137/1.9781611973754.3.

[3] Simon Gog and Matthias Petri. "Compact Indexes for Flexible Top-k Retrieval". In: *CoRR* abs/1406.3170 (2014). arXiv: 1406.3170. URL: http://arxiv.org/abs/1406.3170.

[4] Wing-Kai Hon et al. "Space-Efficient Frameworks for Top-k String Retrieval". In: *J. ACM* 61.2 (Apr. 2014), 9:1–9:36. ISSN: 0004-5411. DOI: 10.1145/2590774. URL: http://doi.acm.org/10.1145/2590774.

[5] 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.

[6] 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.

[7] S. Muthukrishnan. "Efficient Algorithms for Document Retrieval Problems". In: *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '02. San Francisco, California: Society for Industrial and Applied Mathematics, 2002, pp. 657–666. ISBN: 0-89871-513-X. URL: http://dl.acm.org/citation.cfm?id=545381.545469.

[8] Gonzalo Navarro and Yakov Nekrich. "Top-k Document Retrieval in Optimal Time and Linear Space". In: *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '12. Kyoto, Japan: Society for Industrial and Applied Mathematics, 2012, pp. 1066–1077. URL: http://dl.acm.org/citation.cfm?id=2095116.2095200.

[9] Kunihiko Sadakane. "Succinct data structures for flexible text retrieval systems". In: *Journal of Discrete Algorithms* 5.1 (2007), pp. 12–22. ISSN: 1570-8667. DOI: https://doi.org/10.1016/j.jda.2006.03.011. URL: http://www.sciencedirect.com/science/article/pii/S1570866706000141.

[10] 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.

[11] 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.