Agenda

These are the notes from the book **Bioinformatics Algorithms** by Enno Ohlebusch I took for my dissertation exam. The main idea behind this is that first I read whatever piece of text, then wait a few days and only then I take notes, using only my own memory. If it was a good idea I'll know in about a month.

Chapter 1 - Biological Sequences

The complete DNA sequence of an organism is called a *genome*. Not every single nucleotide in a an individual's genome influences the functionality of a given organism. The functional segments of DNA are called *genes*.

Although the individuals in a population of organisms of the same species share most of their DNA, there are positions at which the sequences vary. Most of this genetic variability is caused by single nucleotide variants (SNVs), which stands for a change in a single character in some context, e.g between `ACGA` and `ACTA`. However, larger structural variants are also commonly observed, such as insertions (`AC` $\rightarrow$ `AAAAAAAAC`), deletions (`AAAAAAAAC` $\rightarrow$ `AC`), tandem repeats (adjacent copies of a substring, e.g. `ACGT` $\rightarrow$ `ACGTACGT`) and so on [9]. A special type of SNV is a single nucleotide polymorphism (SNP), which is a change in a single nucleotide that is present with a certain frequency in a population.

Chapter 2 - Exact String Matching

The problem is defined as follows: we have a string S of length n and a string P of length m (called a *pattern*), and we need to find all the occurrences of P in S. Formally, this is the set $\text{Occ} = \{i \mid S[i+m-1] = P\}$. Ofc. brute-force gives us a $O(nm)$ algorithm, but as we can see later, we can do better.

Boyer-Moore-Horspool

The Boyer-Moore-Horspool (BMH) algorithm is based on the two heuristics called *good suffix* and *bad character*. The bad character heuristic causes the algorithm to shift the comparison at least one position for each mismatch. While the BMH still attains the $O(nm)$ worst case running time, it is sublinear in expectation. There is also an extension of the BMH that runs in $O(n+m)$ time (not explained in the book).

The BMH algorithm compares the pattern to each position in S from right to left. The idea behind the bad char. heuristic is that when a mismach happens on the position $T[k] = P[j]$, then we can shift P so that the character $T[k]$ is aligned to the highest index $l \in \{1, \ldots, j-1\}$ such that $P[l] = T[k]$. We can easily see that this shift is *safe*, meaning that we do not skip occurrences of P. As shift P by at least one positoin, oftentimes even by more than one, we save a certain number of comparisons. This number is reasonably large in practice.

```
i = 0
while i < len(S):
    for j in range(len(P)-1, 0, -1):
        if P[i] != S[i]:
            i += bad_char(j)
```

```
            break
    output(i)
```

Knuth-Morris-Pratt (KMP)

The KMP algorithm compares P to each position in S in a left-to-right fashion. When a mismatch occurs at some index $S[i+j] \neq P[j]$, P shifted by k positions to the left, where k is the length of the longest proper prefix of P that is a suffix of $P[1..j-1]$ and simulteneously $P[k+1] = P[j]$. For now, let us assume that we have the k value for each position $j \in [1, m]$ precomputed in an array T. Then the matching is done as follows:

```
i = 0
j = 0
while i < range(len(S)-len(P)-1):
    while j < range(len(P)) and P[j] == S[i]:
        i += 1
        j += 1
    if j == len(P):
        output(j)
    while j > 0 and P[T[j]+1] != S[i]:
        j = T[j]
```

But how do we compute the array T? Let us take a position $P[i]$. The task si to find the largest j, such that $P[1..j]$ is the largest proper prefix of P that is also a suffix of $P[1..i]$. This is, however, almost the same task if we assume that the values $T[k]$ are computed for $k \in [1, i-1]$. Then we are just looking for the largest index k, such that $P[T[k]] = P[i]$.

```
T[0] = 0
for i in range(1, len(P)):
    j = i
    while j > 0 and P[T[j]] != P[j]:
        j = T[j]
    T[i] = j + (P[T[j]] == P[j])
```

Since $T[k]$ is a *proper* prefix, it holds that $T[k] > T[T[k]] > ...$, hence at each position $S[i]$ during the comparison we either increment i or shift the parent P. This results in a $O(n+m)$ worst case running time, which is also optimal.

Aho-Corasick

What if we want to match a set of patterns (a pattern set) $\mathcal{P} = P_1, \ldots, P_k$ of respective lengths $n1, \ldots, n_k$? We can of course run KMP for each P_i giving us a $O(kn+m)$ running time, where $m = \sum_i^k n_k$. We can do better than that with smarter preprocessing. Let us construct a trie of all the patterns $P_1, \ldots, P_k$. This can be done in $O(m)$ time by sequentially insterting the patterns into the intermediate trie, following the 'matching path' for as long as possible and branching off (only once) when necessary. We also store the index of the pattern into the corresponding vertex.

At this point we need to introduce some notation. We will denote $\lambda(e)$ to be the edge label and $\phi(v)$ be the label of a vertex v, i.e. the string obtained by concatenating edge labels on the path from the root to v. Once we have constructed the trie T, we will need the so called *failure links* that we show how to compute later. A failure link $f(v)$ of a node v is the node w such that $\phi(w)$ is the longest

proper prefix of $\phi(v)$ that is also a suffix of $\phi(v)$. It is almost the same thing we precompute for the KMP algorithm.

We are now almost ready to match the pattern set against a string S in the KMP fashion - we simultaneously traverse T and follow the failure links whenever a mismatch occurs or whenever we have successfully found an occurrence of a pattern from $\mathcal{P}$. however, we need to resolve one caveat. If a pattern P_i is a proper substring of a pattern P_j, then P_i can be missed during the matching. This can be fixed by introducing an *output set*, which is defined as the set of pattern indices that can be picked up from either of the $v, f(v), f(f(v)), \ldots, f^d(v) =$ root and denote this set as output(v). Then we report a match to all of the patterns in output(v) whenever we successfully stop the matching at v.

- ☐ example of a missed pattern?

Computation of Failure Links

Naturally, we inquire about how to efficiently construct the failure links and the output sets. Luckily for us, both can be computed simultaneously, effectively and using a simple approach. We proceed iteratively from tyhe root to the leaves in 'levels', i.e. first we process the root, then all the vertices with the depth 1 and so on. During the computation we use the fact that when we process a vertex v, its parent vertex v' with (obviously) smaller depth than v has already been processed and so we know the vertex $f(v') = w$. We can therefore procees similarly as in the KMP preprocessing, following the failure links until we find a suitable vertex.

```
v.f = root for v in vertices # just initialization, will change later
vertices = [1 .. max_depth] # vertices[k] stores all v s.t. d(v) = k
for level in range(1, max_level): # the root link is correctly set
    for v in vertices[level]:
        u = v
        s = label[v]
        while u != root:
            if there is an edge (u, w) labeled by s[-1]:
                v.f = w
```

It is not that obvious, but an amortized analysis (very similar to that for KMP for one pattern P_i) on the total increase resp. decrease of the depth during the computation yields that we can compute both the failure links and the output sets in $O(m)$ time. When we have already done the preprocessing, the matching itself can be done in $O(n)$ time.

Chapter 3 - Range Minimum Queries (RMQs)

Given an array A of n elements from a totally ordered set, we would like to be able to efficiently support the **r**ange **m**inimum **q**ueries $\text{RMQ}(i, j) = k$, such that $A[k] = \min\{A[o] \mid i \leq o \leq j\}$. Moreover, consider a tree T of n vertices, in which we would like to support the **l**owest **c**ommon **a**ncestor queries $\text{LCA}(u, v)$, which gives us the lowest common ancestor w of vertices u, v. We can, of course, solve both problems without any sorts of preprocessing, which results in a $O(n)$ time complexity per query and $O(1)$ of required memory.

This is, however, far from satisfactory as we can most of the times afford to spend some time on preprocessing. It is also possible to simply precompute everything, i.e. spending $O(n^2)$ time to store

all of the queries into $O(n^2)$ words of memory, which can later be accessed in $O(1)$ per query. This stops being feasible rather quickly as n rises. Surprisingly though, we will show that we can, in fact, achieve $O(1)$ time per query with only $O(n)$ preprocessing for each of the problems! Moreover, we will show that the RMQ problem is equivalent to the LCA problem in the sense that one can be effectively solved provided an effective algorithm for the other.

RMQ vs. LCA

Here we show the equivalence of the RMQ and the LCA problems. In each of the implications we assume that there is a $O(n)$-preprocessing/$O(1)$-query time algorithm for one problem and we have the task to devise asymptotically the same method for the other.

RMQ $\implies$ LCA. We do the Euler tour resp. preorder traversal of T:

1. Visit the root
2. Traverse the left subtree
3. Traverse the right subtree

During the traversal we store each vertex identifier into the array $E[1..2n-1]$, and its depth into $D[1..2n-1]$. Afterwards we compute the array $R[1..n]$ which stores the first index of a vertex in E. It is not hard to see that $\mathrm{LCA}(u,v) = E[\mathrm{RMQ}_D(R[u], R[v])]$.

LCA $\implies$ RMQ. We first introduce some notation. The *Cartesian tree* of an array $A[l..r]$ is a tree of which root corresponds to the index $l \leq m \leq r$ such that $A[m]$ is the minimum in $A[l..r]$, its left child is $A[l..m-1]$ if $l < m-1$, otherwise there is no child and similarly $A[m+1..r]$ is the right child if $m+1 < r$. The Cartesian tree of an array A is denoted as $\mathcal{C}(A)$.

Its not hard to notice that a Cartesian tree is not unique (which minimum we take as the root if there is more than one?). To fix this issue we introduce the so called *canonical* Cartesian trees, in which always the leftmost minimum is picked as the root. A canonical Cartesian tree is denoted as $\mathcal{C}^{\mathrm{can}}(A)$ and can be constructed in $O(n)$ time. Before we dig into showing how, we introduce one more notion – the *rightmost path* of the canonical Cartesian tree is the sequence of vertices $\mathrm{root} = v_1, \ldots, v_k$ obtained by starting in the root node and following the edges to the right children while there is a right child. We show how $\mathcal{C}^{\mathrm{can}}(A)$ can be constructed for all the blocks in $O(n)$ time. We can construct a $\mathcal{C}^{\mathrm{can}}(A)$ for an array $A[1..n]$ in $O(n)$ time by the following iterative algorithm:

1. For $i = 1$, the $\mathcal{C}^{\mathrm{can}}(A[1])$ consists just of a root node.
2. For $i > 1$, the $\mathcal{C}^{\mathrm{can}}(A[1..i+1])$ can be constructed from the $\mathcal{C}^{\mathrm{can}}(A[1..i])$ using by means of the following observations. We know that the node representing $i+1$ has to be on the end of the rightmost path in the tree. We therefore start jumping from the last added vertex i upwards until we find a suitable position for the insertion of $i+1$. This is one of the following cases:
   (a) $i+1$ is the right child of i
   (b) $i+1$ is the new root
   (c) $i+1$ is inserted 'somewhere in the middle of the rightmost path'

The algorithm can then be summarized as:

1. Build $\mathcal{C}^{\mathrm{can}}(A[1..n])$ in $O(n)$ time.

2. Prepare $\mathcal{C}^{\text{can}}(A[1..n])$ for constant time LCA queries, which takes $O(n)$ time.
3. Each RMQ(i, j) can be answered as LCA(i, j).

RMQ - The Sparse Table Algorithm

We can achieve a better trade off in preprocessing vs. query time if we do the following. We precompute the RMQ values for all intervals of sizes that are powers of 2. This can be done in $O(n \log n)$ time in using simple dynamic programming, storing the results in a table of $O(n \log n)$ entries. Then every query RMQ(i, j), where $m = i - j + 1$ can be answered by combining two intervals of size 2^k, where $k = \max\{i \mid = 2^i \leq m\}$.

RMQ - The Optimal Algorithm

1. Divide the input array $A[1..n]$ into block of size $s = \frac{\log n}{4}$.
2. Create arrays A' and B', both of size $\frac{n}{s}$, where $A'[i]$ stores the the minimum of the i-th block and $B'[i]$ is the position of that minimum in the block.
3. Preprocess A' and B' as in the sparse M' table algorithm.
4. For each block, precompute the RMQs for all intervals completely contained inside the block.

We can now answer the RMQ(i, j) queries as follows:

1. If $i - j + 1 > s$, then the interval $[i, j]$ spans more then one block. We therefore divide the query into three queries $a = \text{RMQ}(i, k_1), b = \text{RMQ}(k_1, k_2), c = \text{RMQ}(k_2, j)$, where k_1 is the nearest block boundary following i and k_2 is the nearest block boundary preceding j. The values a, c can be obtained from the precomputed values, the value b from the sparse table M' of the block minimums in A'. Of course, we need to remap the index of b, but that is easy. The result of the query is the $\min\{a, b, c\}$.
2. If $i - j + 1 \leq s$, then the interval $[i, j]$ is completely contained within a block, so we just output the minimum according to the precomputed table.

Step 3 requires $O(\frac{n}{s} \log \frac{n}{s}) = O(\frac{4n}{\log n} \log \frac{4n}{\log n}) = O(n)$ time and memory. To estimate the size of the table of all in-block queries, we use the observation that for blocks A, B, it holds that $\text{RMQ}_A(i, j) = \text{RMQ}_B(i, j)$ for all $1 \leq i \leq j \leq s$ if and only if $\mathcal{C}^{\text{can}(A)} = \mathcal{C}^{\text{can}(B)}$. There is $C_n = \frac{1}{n+1}\binom{2n}{n}$ different Cartesian trees[1], which can be bounded by the Stirling's formula as $C_n = O\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$. The table that stores the precomputed queries for all blocks then takes

$$O(C_s \cdot s \cdot s) = O(\frac{4^s s^2}{s^{\frac{3}{2}}}) = O(4^s \sqrt{s}) = O(n)$$

of space.

The last question we need to answer is how to effectively find the type of each block.

We associate the sequence of integers $l_1, \ldots, l_n$ with the aforementioned construction in the following way – l_i is the number of vertices that is on the rightmost path in $\mathcal{C}^{\text{can}}(A[1..i])$ but not on the rightmost path in $\mathcal{C}^{\text{can}}(A[1..i + 1])$. It can be shown that this sequence uniquely characterizes

[1] C_n is known as the *Catalan* number

$\mathcal{C}^{\text{can}}(A[i..n])$. We stress that it is possible to compute the sequence $l_1, \ldots, l_n$ even without explicitly computing the $\mathcal{C}^{\text{can}}(A)$ (an exercise for the attentive reader :D).

Rank and Select in $O(1)$ time for bitvectors (not in the book)

Rank

First we divide the bitvector B of size $|B| = n$ into *superblocks* of size $\log^2 n$ and for each ending position in a superblock we calculate the rank. Since we have $\frac{n}{\log^2 n}$ superblocks, this is takes $O(\frac{n}{\log n}) = o(n)$ bits in total.

Now, each superblocks is further divided into *blocks* of size $\frac{1}{2}\log n$. For each block, the rank *in terms of its parent superblock* is calculated. We have $O(\frac{n}{\log n})$ blocks and each such rank takes $\log\log^2 n = 2\log\log n$ bits, so in total this takes $O(\frac{n \log\log n}{\log n}) = o(n)$ bits.

The last thing we need is to precompute the rank for each bitvector of length $\frac{1}{2}\log n$. There is $2^{\frac{1}{2}\log n} = \sqrt{n}$ such bitvectors, so storing the results in a table takes $\sqrt{n}\log n \log\log n = o(n)$.

Now we have all the ingredients. To get $\text{rank}_1(B, k)$, we first get the rank up to the nearest superblock, then up to the nearest block and finally we look into the table and add what is left to calculate. Obviosuly we only use $o(n)$ bits in addition to the n bits of B.

However, it's not the fastest solution in practice, since we have 3 cache misses in the worst case. To alleviate this at lieast a bit we can use the *popcnt* instruction to compute the rank inside of a block.

Select

First we calculate $\text{select}_b(B, k)$ for each multiple $t_1 = \log n \log\log n$. This divides B into *superblocks* or different sizes. We can then classify the superblocks into two categories:

1. Large and sparse superblocks of which size is $\geq t_1^2$. For these we can afford to directly precompute and store all the selects. Together it takes $O(\frac{n}{t_1^2} t_1 \log n) = o(n)$ bits.
2. Smaller and dense superblocks of size $< t_1^2$. For these we use the same approach for $t_2 = (\log\log n)^2$. For the multiples of t_2 we precompute and store the selects, which takes $O(\frac{n}{t_1^2}\log\log n) = o(n)$ bits total, since the select for one position inside of a superblock takes $\log\left((\log n \log\log n)^2\right) = O(\log\log n)$ bits.
    1. For the small (size $\geq t_2^2$) superblocks we directly write the positions of ones (inside of outer). Each such position takes $\log t_1 = O(\log\log n)$ bits, so in total this takes $O(\frac{n}{t_w^2} t_1 \log\log n) = O(\frac{n}{(\log\log n)^4}(\log\log n)^2 \log\log n) = O(\frac{n}{\log\log n}) = o(n)$ bits.
    2. There is only a small number of small superblocks (size $\leq t_2^2$), so we can precompute and store them into a table of size $\frac{n}{t_2^2} 2^{(\log\log n)^4} = o(n)$.

AD1: No known relation between $\text{select}_0(B, k)$ and $\text{select}_1(B, k)$.

AD2: In practice, binary search using rank may suffice.

souce: [Kuko's notes] (https://people.ksp.sk/~kuko//ds/mat/succinct.pdf)

Chapter 4 – SA, LCP

Suffix Arrays and LCP Arrays

Suffix trees [12] are among the most powerful and versatile string data structures, enabling to solve a variety of problems in asymptotically optimal time, but their memory requirements – 20 bytes per input character in the worst case [1] – make them impractical for many tasks. Manben and Myers [8] addressed this issue by introducing suffix arrays (SAs), a data structure that is more space-efficient in practice, but can be also leveraged to a large variety of problems on strings.

The suffix array SA_T of a string T of length n is a permutation of $[1, n]$ for which it holds that $SA_T[i]$ is the starting position of the i-th suffix of T in terms of the lexicographic order. For example, if we have $T = \texttt{AGG\$}$, then $SA_T = [4, 1, 3, 2]$, because $\$ \prec AGG\$ \prec G\$ \prec GG\$$. SAs therefore asymptotically occupy $O(n \log n)$ bits of space, which is the same as suffix trees, but in practice only around 4 bytes per input character are used [1]. SAs can be computed in optimal $O(n)$ time by a large variety of different algorithms that make trade-offs between time and memory usage as well as the achievable degree of parallelism, e.g. the popular *Skew* [7] or *divsufsort* [6] algorithms.

Induced Sorting

The basic idea of induced sorting (IS) algorithm is that we only need to sort a subset of the suffixes, called LMS suffixes, which in turn can be extended into the order of all of the suffixes. Let us label each suffix S_i by either L in case $S_i > S_{i+1}$ or S in case $S_i < S_{i+1}$ (S is \$-terminated, so we cannot have equality) and store the label for the suffix S_i in the *type array* $T[i]$. This classification can be done in $O(n)$ time using the fact that \$ is S-type and the following the case distinction:

1. if $S[i] = S[i + 1]$, then $T[i] = T[i + 1]$,
2. if $S[i] < S[i + 1]$, then $T[i] = S$,
3. if $S[i] > S[i + 1]$, then $T[i] = L$.

We furthermore call a suffix S_i a LMS suffix (**L**eft**M**ost**S**maller suffix) when (i) it is of type S and (ii) $i = 1$ or S_{i-1} is L type. In other words, the LMS suffixes are the S suffixes that have no S suffix as a left neighbor. Let us say we already have sorted the LMS suffixes into their correct relative order by a procedure **mystery()**. We first show how we can use this information to sort the rest of the suffixes and then how to actually implement **mystery()**. Before we show how we can sort the rest of the suffixes, we first mention one more important observation, which is that if we have suffixes S_i and S_j such that $T(i) = S$ and $T[j] = L$ and both S_i and S_j start with the same character, then $S_j < S_i$. We can see this letting $S_i = uavcw, S_j = uvabw$ for $v, w \in \Sigma^*$ and $a, b, c \in \Sigma$, and distinguishing the two following cases:

1. If v consists solely of a's, then from $S_i < S_{i+1}$ we get $c > b$. Conversely, from $S_j > S_{j+1}$ we get $c < b$, which is a contradiction.
2. Let d be the first character in v such that $d \neq a$. Then similarly we get a contradiction from $a > d$ and $d < a$.

Now we are ready to explain the core structure of the algorithm. First we create an array $A[1..n]$ that will serve as a placeholder for the indices of the sorted suffixes and that is conceptually divided into *buckets*. Each such bucket represents a character $c \in \Sigma$, its size corresponds to $\mathrm{Occ}(c, S)$ and the buckets are sorted according to the order of Σ, so the first bucket correspond to \$ and has a size

one, etc. In fact, we can access the beginning of a bucket of a character c by $C[c]+1$ and its end by $C[c+1]$. Initially, each entry $A[i]$ is either filled with $\perp$ (undefined value) or $A[i]=j \neq \perp$ which $A[i] \neq \perp \neq A[j]$, $i<j$ means that $S_{A[i]} < S_{A[j]}$. Initially, of course, the array contains the relative ranks of the LMS suffixes. Then we do the following:

1. *Phase 1*: Sort the LMS suffixes using the **mystery()** function.
2. *Phase 2*: Do the following:
   (a) Iterating over the sorted LMS suffixes from right to left, for each suffix position i, add it to the current end of the $S[i]$-bucket and shift the end of the bucket one position to the left.
   (b) Going from left to right, for each $A[i] \neq \perp$ s.t. S_i if of type L, we set $A[i-1] = A[i]-1$. In this step we correctly sort all the L-type suffixes.
   (c) Going from right to left, for each $A[i] \neq \perp$ s.t. S_i if of type S, we set $A[i-1] = A[i]-1$. In this step we correctly sort all the S-type suffixes.

It can be shown that the step 2b resp. 2c of the Phase 2 correctly sorts the L resp. S suffixes.

Now about the **mystery()**. Recall that **mystery()** sorts to LMS suffixes. We will be needing some additional notation. We call a LMS substring a substring of S that both starts and ends at a LMS position. and it does it accordingly:

1. Initialize an array A of the same function as in phase 2 and an array $P[1..m]$ in which m is number of LMS positions in S.
2. Iterating over the string S from right to left, if the position i is a LMS position, then add it to the current end of the $S[i]$-bucket and shift the end of the bucket one position to the left. Moreover, add the position i to the current end of the P array and move the end to the left.
3. Going from left to right, for each $A[i] \neq \perp$ s.t. S_i if of type L, we set $A[i-1] = A[i]-1$.
4. Going from right to left, for each $A[i] \neq \perp$ s.t. S_i if of type S, we set $A[i-1] = A[i]-1$.
5. Initialize variables $\overline{i}=1, prev=1$ and the array $LN[1..m]$ – the *lexicographic names* of the LMS suffixes. Since \$ is the lexicographically smallest LMS substring, we set $LN[n]=1$. Next we iterate over the remaining suffixes and compare the current suffix S_i with S_{prev} character by character. If they differ, increment $\overline{i}$. We then set $LN[P[i]] := \overline{i}$, $prev := i$ and continue with the next suffix.
6. If $\overline{i}=n$, then we can directly construct the suffix array $\overline{SA}$ of $\overline{S}$, where $\overline{S}[k] = \overline{LN[P[k]]}$ for $k \in [1..m]$.
7. Otherwise construct the suffix array of $\overline{S}$ *recursively.*
8. The resulting suffix array of LMS suffixes is then $P[\overline{SA}[1]], \ldots, P[\overline{SA}[m]]$.

Correctness

Let us first discuss the correctness of steps 2b and 2c of Phase 1, for which the proofs are very similar. In step 2b we first prove that when we place a L suffix, it is placed to the correct position in the order. We do it by induction on the number q of placed L-type suffixes. For $q=0$, no suffix is placed, so the basis holds. Now we prove that if q lexicographically smallest L suffixes have been placed correctly, then the $(q+1)$-th largest suffix S_j will be placed as well. Let $A[i]=j+1$, so at the position i we add the suffix $S_j = cS_{j+1}$ to the current front of the c-bucket. We prove that there is

no suffix $S_k > S_j$ in the bucket. To achieve a contradiction, suppose there is the suffix S_k. From the fact that S_k appears at a sooner position in the c-bucket, there must have been $A[i'] = k+1$ such that $i' < i$. Moreover, since both S_k, S_j are in the c-bucket, we have $S_k = cS_{k+1}$, $S_j = cS_{j+1}$ and from $S_k > S_j$ we also get $S_{k+1} > S_{j+1}$. This is, however, a contradiction with the fact that from the induction hypothesis resp.

Time complexity

We only need to address the recursion factor since other operations clearly take $O(n)$ time. Since the LMS positions are at least one position apart, the recursive step is called on an array of size $m \leq \frac{n}{2}$. Combining this with the Master theorem, we get that the algorithm takes $O(n)$ time.

Longest Common Prefix (LCP) Array

Let $LCP[i] = |\text{lcp}(S_{SA[i]}, S_{SA[i-1]})|$, i.e. the length of the longest common prefix of the lexicographically 'consecutive' suffixes $S_{SA[i]}$ and $S_{SA[i-1]}$. We can compute the LCP array in $O(n)$ time from the SA by means of the following observation: if $SA[i] = k, SA[j] = k+1$ and $m = \text{lcp}(S_{SA[i]}, S_{SA[i-1]})$, then $\text{lcp}(S_{SA[j]}, S_{SA[j-1]}) \geq m-1$, because S_{k-1} is 1 character shorter than S_k. This saves us redundant character comparisons.

We can use this information to build the LCP array in one pass. More precisely, we build something that is called a Permuted LCP (PLCP) array that is defined as $\text{PLCP}[i] = \text{LCP}[\text{ISA}[i]]$ (or alternatively $\text{PLCP}[\text{SA}[i]] = \text{LCP}[i]$) and transform it later to LCP using this equation. So in PLCP, the lcp values are computed for the prefix in the string order rather than their lexicographic order. We therefore traverse through the string S from left to right for each suffix S_i for $i \in [1..n]$ while keeping track of the longest common prefix ℓ from the previous suffix pair $S_{i-1}, S_{\phi(i-1)}$, where ϕ is defined as

$$\phi(i) = \begin{cases} \text{SA}[\text{ISA}[i]-1], & \text{if } i > 1 \\ -1, & \text{otherwise} \end{cases}$$

We then simply compute $\text{PLCP}[i] = |lcp(S_{i+\ell}, S_{\phi(i)+\ell}|$. Before the next iteration, we do $\ell := \max(\ell - 1, 0)$.

If we wanted to construct the LCP array right away, we could use the same principle but we compare the suffix $S_j = S_{\text{ISA}[i]}$ with $S_k = S_{\text{SA}[j-1]}$.

When we want to compute $lcp(S_{\text{SA}[i]}), S_{\text{SA}[j]})$, we can utilize the RMQ(, o)ver the LCP. It is not difficult to see that

$$lcp(S_{\text{SA}}[i], S_{\text{SA}}[j]) = \min_{k \in [i+1,j]} \text{LCP}[k] = \text{RMQ}(i+1, j).$$

Enhanced Suffix Arrays (ESAs)

The basic idea is that ESA = SA + LCP. It is awesome, because we can use it to simulate a suffix tree with a better memory print and better memory locality.

Suffix Tree Simulation

The simulation is based on the concept of lcp interval. To be precise, a *lcp interval* for a lcp value of ℓ – denoted as $\ell - [i..j]$ – is defined as an interval $[i..j]$ of the LCP array, at which:

- $\text{LCP}[i] < \ell$ and $\text{LCP}[j+1] < \ell$,
- $\text{LCP}[k] \geq \ell$ for all values $k \in [i+1..j]$,
- $\text{LCP}[k] = \ell$ for *at least one* $k \in [i+1..j]$.

A special case of this is a singleton interval encompassing a single LCP value, denoted by simply $[i, i]$. Each lcp block $\ell - [i..j]$ is associated with a word ω that is the longest common prefix of the suffixes $S_{i+1}, \ldots, S_j$. Expectedly in this case it holds that $|\omega| = \ell$. The indices $k \in [i+1..j]$ at which $\text{LCP}[k] = \ell$ are called the *ℓ-indices* of the respective lcp interval.

The *child lcp interval* of a lcp interval $\ell - [i..j]$ is the lcp interval $m - [p..q]$, such that:

1. $i \leq p \leq q \leq j$
2. $\ell < m$

We therefore have this implicit tree-like hierarchy in the LCP array, which is tightly linked to the suffix tree of S. The internal nodes of the ST correspond to the non-singleton $\ell - [i, j]$ intervals, while the leafs correspond to the singletons. This implied ST can be traversed by either top-down or bottom-up fashion, which we now explain in more detail a bit later. First we characterize the child resp. parent interval of a particular lcp-interval and describe how we can compute them.

Children interval(s). Let $\ell - [i, j]$ be a lcp interval. Let $p_1, \ldots, p_k$ be the ℓ-indices of $\ell - [i..j]$. The children of $\ell - [i..j]$ are then the intervals $[i, p_1 - 1], [p_1, p_2 - 1], \ldots, [p_{k-1}, p_k - 1], [p_k, j]$.

PSV, NSV and the Parent Interval. To get the parent interval of a particular lcp interval, we need to introduce some additional arrays. For an array A, let $\text{PSV}[i] = \max\{j \mid A[j] < A[i] \text{ and } j < i\}$ and $\text{NSV}[i] = \min\{j \mid A[j] < A[i] \text{ and } j > i\}$. We can calculate both arrays in $O(n)$ time using the following algorithm:

```
PSV = [-1]*(n+1)
NSV = [-1]*(n+1)
s = stack([])
for i in range(1, n):
    while A[i] < A[s.top()]:
        x = s.pop()
        NSV[x] = i
    if A[i] == A[s.top()]:
        PSV[i] = PSV[s.top()]
    else: # A[i] < A[s.top()]:
        PSV[i] = s.top()
    s.push(i)
```

If we have a particular lcp interval $\ell - [i..j]$, we can obtain its parent interval by $[\text{PSV}[k], \text{NSV}[k]]$, where k is a ℓ-index of $[i..j]$. Furthermore, using the following case distinction we get:

1. $\text{LCP}[i] < \text{LCP}[j+1]$, or alternatively $\text{NSV}[i] > j+1$, then the parent interval is $[i, \text{NSV}[j+1]]$ and $[i..j]$ is its first child with $j+1$ being the first $\text{LCP}[j+1]$-index.
2. $\text{LCP}[i] > \text{LCP}[j+1]$, or alternatively $\text{PSV}[j+1] < i$, then the parent interval is $[\text{PSV}[j+1], j]]$ and $[i..j]$ is its last child with i being the last $\text{LCP}[i]$-index.
3. $k = \text{LCP}[i] = \text{LCP}[j+1]$, then i and $j+1$ are two consecutive k-indices of $[i..j]$ and the parent interval is $[\text{PSV}[i], \text{NSV}[i]]$.

Bottom-up Traversal This approach uses a stack and one traversal through the LCP array. Let *top* be the reference to the top of the stack. We maintain the invariant that the top of the stack has its left boundary correctly set.

```
LCP[0..n+1] # LCP[0] = LCP[n+1] = -1
s = stack([LCP[0]])
for i in range(1, len(LCP)):
    while LCP[i] < LCP[s.top()]:
        k = s.pop()
        process(k, i)
    if LCP[i] > LCP[s.top()]:
        s.push(i)
```

We now prove the correctness of the method. Obviously the lcp values on the stack are strictly increasing.

Top-down Traversal This one is based on the observation that we can get generate the children intervals of an lcp interval $\ell - [i..j]$ as follows.

Let us now assume we have arrived to the interval $\ell - [i..j]$ and want to visit its children. To do this, we employ the RMQ in the following way: 1. We generate RMQ(i, j) to get the first ℓ-index of $[i..j]$.

```
k = i
while k < j:
    k = RMQ(k, j)
    process(k, j)
```

Chapter 5 – Applications of ESAs

- ☐ finding repeats

Exact String Matching

This is really easy, similarly to suffix trees, need to find a path from the root that corresponds to the pattern P. The leaves in the ST in the subtree rooted after the match are the suffix numbers / occurrences of P in S. It's similar with ESAs, we se

Lempel-Ziv Factorization

A Lempel-Ziv (LZ) factorization of a string S is a

Finding Repeats

Two (or more) occurrences of a substring ω in S are called *repeats*. We classify reads as:

1. *Longest* – repeats of the largest length, i.e. such that no other repeat is longer.
2. *Maximal* – repeats which if we extend to either side the number of occurrences decreases.

3. *Supermaximal* – repeats which if we extend to either side it stops being a repeat (i.e. the number of drops to 0). Or equivalently, a supermaximal repeat is a repeat that is not a proper substring of another repeat. Each supermaximal repeat is a maximal repeat.

4. *Repeated pairs* – repeats in the form (p_1, p_2, ℓ), where p_1 and p_2 with $p_1 < p_2$ are the starting positions of the occurrences and ℓ is the length of the repeat. Repeated pairs are said to be *left maximal* (*right maximal*) if they cannot be extended to the left (right). Repeated pairs are maximal if they are both left and right maximal.

5. *Tandem arrays* – repeats in the form of $\omega = u^k$ for $k \geq 2$, i.e. occurrences of which appear consecutively. Specifically, tandem arrays in the form uu (i.e. $k = 2$) are called *tandem repeats*.

6. *Periodicities* – repeats that can be written in the form $w = v^k u$, where $k \geq 2$ and u is a proper prefix of v. Naturally we can also define left maximal, right maximal and maximal periodicities.

Longest Repeats

Very easy, all longest repeats have length $m = \max\{\text{LCP}[i] \mid i \in [1..n]\}$. A repeat of length m is of course detected by $\text{LCP}[i] = m$. To avoid duplicates in the output, we only output the last entries of each 'local maximum' in the LCP. We therefore iterate over the LCP array and output the positions from of the SA that correspond to the repeats of length m.

```
m = max(LCP)
longest_reps = []
for i in range(n+1):
    # to avoid duplicate outputs
    if (LCP[i] == m) and (LCP[i+1] < m):
        longest_reps.append(S[SA[i]:SA[i]+m])
```

However, if we explicitly output the longest repeats as strings, the time complexity of our algorithm grows to $O(n \log n)$, since the total length of all longest repeats can be proportional to $\Theta(n \log n)$ (e.g. the binary de Bruijn string). We could avoid this by only outputting the positions of the longest repeats, which would get us the desired $O(n)$ algorithm.

Supermaximal Repeats

We can detect these by checking the the so called 'local maxima' of the LCP array for right maximality. A local maximum is a an lcp interval $\ell{-}[i, j]$ in which $\text{LCP}[k] = \ell$ for $i{+}1 \geq k \geq j$. In addition to being a local maximum, an interval $[i, j]$ needs to satisfy that the characters $S[\text{SA}[i] - 1], \ldots, S[\text{SA}[j] - 1]$ are pairwise distinct. This check can be accomplished by maintaining a bitvector $B[1..\sigma]$ initialized to all zeros which we will use as an indicator of the preceding characters at positions $[i, j]$ which we have already seen. We will also save a `char_list` of scanned characters to serve when erasing the written ones into B, since scanning the whole B to reinitialize it would add an unnecessary σ-factor to the time complexity of the checking procedure. Clearly, the time complexity of the algorithm is now $O(n)$, which is optimal. A pseudocode can be seen in Alg. 1.

Maximal Repeats

In maximal repeats, we can adopt a similar approach as in supermaximal repeats (Alg. 1) since every local maximum in the LCP array is a candidate maximal repeat, however, we have to change

```
char_list = []
B = [0]*len(C)
supermaximals = []

for i in range(1, n+2):
    rb = i
    if LCP[i] > LCP[i-1]:
        lb = i-1
    if LCP[i] < LCP[i-1]:
        # check for left-maximality
        is_ok = True
        for k in range(lb+1, rb):
            c = S[ ( SA[k] - 1) % n]
            if B[c] == 1:
                is_ok = False
                break
            B[c] = 1
            char_list.append(c)
        # reinitialize B to zeros
        for x in list:
            B[x] = 0
        char_list = []
        if is_ok:
            supermaximals.append( (lb, rb-1, LCP[lb+1]) )
```

Figure 1: An algorithm to find all supermaximal repeats using the ESA.

the left-maximality check. In more detail, the condition for the left-maximality in maximal repeats is that the characters preceding the local maximum $\ell - [i, j]$ *must not be all the same.*

This can be accomplished by maintaining a position `last_diff` of the last occurrence of different preceding characters $S[\text{SA}[\texttt{last\_diff} - 1] - 1] \neq S[\text{SA}[\texttt{last\_diff}] - 1]$. If `last_diff` points to the inside of the candidate interval $[i, j]$, i.e. $\texttt{last\_diff} > i$, then we have a maximal repeat, otherwise it is not left-maximal.

Obviously, the Alg. 2 also runs in the optimal $O(n)$ time.

Maximal Repeated Pairs

Here we have to output all the pairs of repeats that cannot be extended to either side. To this end we will utilize the bottom up traversal of the (implicit) LCP interval tree. We define $\mathcal{P}_{[i,j]}$ to be the set of all SA entries in the $[i, j]$ interval. Moreover, for $a \in \Sigma$, let us define

$$\mathcal{P}_{[i,j]}(a) = \{k \mid k \in [i, j], S[\text{SA}[k] - 1] = a\},$$

i.e. the $\mathcal{P}_{[i,j]}(a)$ is the set of all indices in $\mathcal{P}_{[i,j]}$ that are preceded by the character a in S.

The computation of all maximal repeated pairs works as follows. We will traverse the lcp interval tree in the bottom up fashion and for each lcp interval $\ell - [lb, rb]$ we can assume that its children intervals have already been processed. We use the observation that for any $k \in P_{[i,j]}(a), l \in P_{[i',j']}(b)$ for any

```
last_diff = 0
maximals = []

for i in range(1, n+2):
    rb = i
    if LCP[i] != LCP[i-1]:
        last_diff = i
    if LCP[i] > LCP[i-1]:
        lb = i-1
    if LCP[i] < LCP[i-1]:
       if last_diff > lb:
            maximals.append( (lb, rb-1, LCP[lb+1]) )
```

Figure 2: An algorithm to find all maximal repeats using the ESA.

two different children intervals $[i, j]$ and $[i', j']$ and $a \neq b$ it holds that (k, l, ℓ) is a maximal repeated pair. This is due to the fact that the left-maximality is guaranteed by $a \neq b$ and right-maximality by $[i, j] \neq [i', j']$.

For every $a \in \Sigma$ we therefore define $P^q_{[i,j]}(a)$ to be the intermediate value of $P_{[i,j]}(a)$ after processing the first q children. For the $(q+1)$-th child $[i', j']$ we first output all maximal repeated pairs (k, l, ℓ) for $k \in P_{[i,j]}(a), l \in [i', j']$ such that $S[\mathrm{SA}[l] - 1] \neq a$. Subsequently we do an update:

$$P^{q+1}_{[i,j]}(a) = P^q_{[i,j]}(a) \cup \{k \mid k \in [i', j'], S[\mathrm{SA}[k] - 1] = a\}.$$

The total running time of this algorithm is therefore $O(n + z)$, where z is the number of all maximal repeated pairs.

Periodicities

We denote a periodicity by (b, e, ℓ), such that $S[b..e] = v^k u$ and $|v| = \ell, |u| < \ell$.

We can find all the periodicities of in a string S of length n in $O(n)$ time. The basis of this algorithm is the Lempel-Ziv factorization. We divide the set of periodicities into type 2 – the periodicities that are completely contained in LZ phrases and type 1 – all the other periodicities. The high level overview of the algorithm is:

1. Compute the LZ factorization of S.
2. Compute the periodicities of type 1.
3. Compute the periodicities of type 2.

Before we start explaining the asymptotically optimal algorithm, we first mention some important notions. The first one is that the substring $S[b..e]$ is a periodicity of a period ℓ if and only if $S[i] = S[i + \ell]$ for $b \leq i \leq e - \ell$. The proof of this fact quite simple and the statement is obvious. The second thing is that we say that a periodicity (b, e, ℓ) has a *period to the left* of position i if it has the ℓ-length substring ending at the position $i - 1$, and *period starting at* position i if it has the ℓ-length substring starting at i.

We will first explain how we can find the type 1 periodicities. Let $s_1, \ldots, s_m$ be the LZ factorization of string S of length n. The solution is based on the characterization of type 1 periodicities by

dividing them into disjoint groups. In particular, a periodicity (b, e, ℓ) that ends within a factor $s_j = S[b_j..e_j]$ can either:

1. Have either a period to the left of b_j or starting at b_j (or both), which we further divide into:
   (a) Those that have a period to the left of b_j.
   (b) Those that have a period starting at b_j but have no period to the left of b_j. In this case we have $\ell \leq |s_j|$.
2. Those that start at $b_j \geq b$ and end at $e = e_j$. In this case we have $\ell \leq |s_j|$.
3. Those that start at $b_j = b$ and end at $e \leq e_j$. In this case we have $\ell \leq |s_j|$.

```
# functions lcs (lcp) computes the length of the LCS (LCP) in O(1) time
B = [0..m-1] # B[i] is the beginning index of the i-th phrase
E = [0..m-1] # E[i] is the ending index of the i-th phrase
periodicities = [] # this will be the output
    for i in range(1, m-1):
        for l in range(1, min(E[i-1], len(S[j-1]) + len(S[j]))):
            # check for periodicities that have period to the left of i
            L = lcs(E[i-1], E[i-1]-l)
            R = lcp(B[i], B[i]-l)
            if (L+R >= l) and (R > 1 or E[i-1] - l - L > B[i-1]):
                periodicities.append(E[i-1] - l - L, E[i-1] + R)
        for l in range(1, S[i]+1):
            # check for periodicities that have a period starting at i
            L = lcs(E[i-1], E[i-1] + l)
            R = lcp(B[i], B[i] + l)
            if (L+R >= l) and (L < l) and (B[i] + l + R <= E[i]):
                periodicities.append(E[i-1] - L, B[i] + l + R)
```

Figure 3: Computation of type 1 periodicities.

So we know that periodicities that start and end within s_j, or at least have a period starting at b_j but no period to the left of b_j, have their periods upper bounded as $\ell \leq |s_j|$. Interestingly, it is also possible to find an upper bound of the periodicities that have a period to the left of b_j.

It is the consequence of the fact that if (b, e, ℓ) has a periodicity to the left of b_j, the following holds:

1. $e \leq e_j$
2. If $e_{j-1} < e$, then (b, e, ℓ) does not have a period to the left of b_{j-1}
3. If $e_{j-1} < e$ then it holds that $|S[b..e]| < 2|s_{j-1}s_j|$ (so $\ell < |s_{j-1}s_j|$)

Let us now sketch the proof of these observations.

1. If it was the case that $e_j < e$, then $S[b_j - \ell..e - \ell]$ is an occurrence of $S[b_j..e]$ that starts at an earlier position and $|S[b_j..e]| > |[b_j..e_j]|$, which is a contradiction with the way the LZ parse is constructed, hence $e \leq e_j$.
2. Now if (b, e, ℓ) had a period to the left of b_{j-1}, then by 1 we would get $e \leq e_{j-1}$, which is a contradiction.

3. Finally, suppose $|S[b..e]| \geq 2|s_{j-1}s_j|$, so by $e \leq e_j$ at least half the characters of $S[b..e]$ appear *strictly before* the position b_{j-1}. Since $|S[b..e]| \geq 2\ell$, or in other words $\ell \leq \frac{1}{2}|S[b..e]|$, then that also means there is a period to the left of b_{j-1}, which contradicts 2.

Now that we can bound the period of every periodicity of type 1 that ends in a particular phrase, we are ready to propose algorithm to compute them in 3. Let us argue that the presented algorithm computes all type 1 periodicities. Let us now prove its correctness. We claim that for each phrase s_j for $j \in [2, m]$, the proposed algorithm will *exactly once* output every periodicity (b, e, ℓ) such that it:

1. ends within s_j and crosses the border to the left neighbor. Here we further divide the following cases:
   (a) Those that have a period to the left of b_j.
   (b) Those that have a period starting at b_j but have no period to the left of b_j.
2. starts at $b \geq b_j$ and end at $e = e_j$.
3. starts at $b = b_{j-1}$ and end at $e \leq e_{j-1}$.

The periodicities which have a period to the left of the beginning b_j of the current phrase s_j are being handled in the first for-loop. The first condition `R > 1` in the 'or part' of the if-statement guarantees that only those periodicities ending in s_j will be output (we know that if the periodicity continues to s_j, then $e \leq e_j$ in this case), while the second guarantees that only those that begin after b_{j-1} and end at e_{j-1} will be output. The second for-loop handles the periodicities that that have periods starting at the particular position b_j. We must therefore check if there is no period to the left of b_j by `L < l` and also if the period ends within s_j by `B[i] + l + R <= E[i]` to avoid duplicate outputs. This proves the claim.

Let us now analyze the time complexity of computing type 1 periodicities. Since *lcp* and *lcs* can be answered in $O(1)$ time, the question is how many times do we ask these queries. That is the number of the for loop iterations over all phrases, which is

$$\sum_{j=2}^{m} |s_{j-1}s_j| - 1 + |s_j| = 2n + n = O(n).$$

Let us now address the problem of computing type 2 periodicities. We do this by using the observation that if a periodicity (b, e, ℓ) is properly contained in a phrase s_j, i.e. $b_j < b < e < e_j$ (which also implies $|s_j| > 4$), then there is a previous occurrence $S[b - \delta_j..e - \delta_j]$ of it, where $\delta_j = b_j - t_j$, where t_j is the start of the previous occurrence of s_j.

So we are basically looking for re-occurrences of type 1 periodicities that are completely contained in the phrases. Let us have the output of Alg. 3 in the form of n lists in which $L[i]$ is the list containing the periodicities that end at the position i. We will use counting sort to transform these lists so that $L[i]$ contains all the periodicities *starting* at position i in increasing order of their ending positions. This process takes $O(n)$ time, since there are at most $O(n)$ periodicities of type 1 (which follows from the time complexity of Alg. 3.). The lists $L[i]$ are the processed as in Alg. 4.

Alg. 4 maintains the invariant that after finishing the search for the phrase s_i, it holds that for $b_j \leq j \leq e_j$ $L[j]$ stores all the periodicities starting at j in the increasing order of their ending positions. This is because before adding anything new into $L[j]$ they were already sorted according to their ending positions and each periodicity in $L[j]$ was ending *after* e_j. We then add new periodicities

in such a way that they remain sorted and each of the newly added periodicities ends *before* e_j. This proves the claim that Alg. 4 finds all periodicities of type 2.

The time complexity of Alg. 4 is clearly $O(n+z)$, where z is the number of all maximal periodicities of type 2 in S. This follows from the fact that each time the innermost for-loop(s) is executed, we have found a new periodicity.

```
prev_occ[0..n-1] # previous occurrence of the i-th phrase (-1 stands for no prev. occ.)
B = [0..m-1] # B[i] is the beginning index of the i-th phrase
E = [0..m-1] # E[i] is the ending index of the i-th phrase
L = [0..n-1] # L[i] is the list of periodicities (b, e, l) starting at position i
for i in range(1, n):
    if prev_occ[i] == -1: continue
    di = i - prev_occ[i]
    for j in range(B[i]+1, E[i]):
        periodicities = []
        for (b,e,l) in L[j-di]:
            if j + (b - e + 1) >= E[i]: break # further periodicities are only longer
            periodicities.append( (b+di, e+di, l) )
        for p in reversed(periodicities):
            L[j].prepend(p)
```

Figure 4: Computation of type 2 periodicities.

MUMs and MEMs

In this section we focus on two definitions of exact matches that are useful in whole genome comparison. Let S_1, S_2 be strings of lengths n_1, n_2. Then we define:

1. *Maximal Exact Matches (MEMs)*: An occurrence of a substring $S_1[i \dots j]$ at position k in S_2 is a maximal exact match (MEM) if and only if the following conditions hold:
   (a) $S_1[i \dots j] = S_2[k \dots k + (j-i)]$,
   (b) Either $i = 1$ or $k = 1$ or $S_1[i-1 \dots j] \neq S_2[k-1 \dots k + (j-i)]$.
   (c) Either $j = n_1$ or $k + (j-1) = n_2$ or $S_1[i \dots j+1] \neq S_2[k \dots k + (j-i) + 1]$.
   This query returns the set $MEMs := \{(i,j,k) \mid S_1[i \dots j]$ is a MEM at position k in $S_2\}$.
2. *Maximal Unique Matches (MUMs)*: an occurrence of a substring $S_1[i \dots j]$ at position k in S_2 is a *maximal unique match* if it is a MEM and simultaneously occurs only once in S_1 and once in S_2. The output of this query is the set $MUMs := \{(i,j,k) \mid S_1[i \dots j]$ is a MUM at position k in $S_2\}$.

We now address the problem of finding all MUMs and MEMs with the help of the ESAs.

MEMs

MEMs between two strings S_1, S_2 are basically just maximal repeated pairs in the form $(k_1, \ell, n_1 + 1 + k_2, \ell)$. Therefore it suffices to find all repeated pairs in the GESA of S_1, S_2 and output only those whose occurrences are in separate copies – something which is easy to check in a GESA.

MUMs

In MUMs we need to find MEMs that are also unique. Observe that these candidate MEMs are adjacent in the GESA of S_1, S_2 and we merely need to check uniqueness and right-maximality. This can be easily implemented as a single pass through the LCP array as in 5.

```
MUMs = []
n_1 = len(S1)
n_2 = len(S2)
for i in range(3, n+1):
    if D[i - 1] != D[i]:
        if LCP[i - 1] < LCP[i] and LCP[i+1] < LCP[i]:
            if S[ (SA[i-1] - 1) % n_1] == S[ (SA[i] - 1) % n_2 ]:
                if D[i] == 0:
                    MUMs.append(SA[i]-1, SA[i-1]-1, LCP[i])
                else:
                    MUMs.append(SA[i-1]-1, SA[i]-1, LCP[i])
```

Figure 5: Computation of all MUMs of strings S_1 and S_2 using their GESA.

Shortest Unique Substrings

In this problem we are looking for all shortest substring of S that start at a position i, for each position i that is unique in S. We call these strings the *shortest unique substrings (SUSs)*. We can do this by comparing each suffix S_i with S_{i-1} and S_{i+1}. It may, however, happen that S_i is a prefix of S_{i+1} (S_{i-1} cannot be a prefix of S_i due to how we sort suffixes), so we need an additional check. If $|S_i| \geq \ell = \max(\text{LCP}[i], \text{LCP}[i+1]) + 1$, then $S[i..i+\ell-1]$ is obviously the shortest unique substring starting at i.

```
LCP = [1..n+1] # LCP[0] = LCP[n] = -1
SUS = [-1]*n
for i in range(1, n):
    if LCP[i-1] < LCP[i] and LCP[i] > LCP[i+1]:
        l = max(LCP[i], LCP[i+1]) + 1
        if (n - SA[i] + 1) >= l:
            SUS[SA[i]] = l
```

String-specific SUSs

Given a collection of strings $S^1, \ldots, S^m$, a string s is called *string-specific* if it is a substring of exactly one of the strings. Moreover, a string s is called *S^k-specific* if it is a string-specific string that is a substring of $S^k, k \in [1, m]$.

We can find the shortest string-specific substring for every $k \in [1, m]$ by the BFS of the GESA of $S^1, \ldots, S^m$. We use an array *minpos* which will store the position of the S^k-specific SUS at $minpos[k]$ and its length in *minlen* (the *minpos* is initialized of -1, the *minlen* to n).

During the traversal, we maintain the invariant that that each w-interval in the queue has $df(w) > 1$ (the document frequency of w). When we pop the interval w-interval $\ell - [i..j]$ from the queue, we inspect all of its child intervals and for each one we distinguish the following cases:

1. we have a singleton interval $[p,p]$ of the substring w. Let us inspect whether the symbol $S^{\#}_{\mathrm{SA}[p]}[\ell+1] = \#_k$, where $j = D[k]$. This is the case iff $\ell = n_k$. We further distinguish:
   (a) If $S^{\#}_{\mathrm{SA}[p]}[\ell+1] = \#_k$ then no prefix $S^{\#}_{\mathrm{SA}[p]}$ ending before $\#_k$ is S^k-specific, so there is nothing to do.
   (b) Otherwise $S^{\#}_{\mathrm{SA}[p]}[\ell+1] \neq \#_k$, so the $\ell+1$ prefix of $S^{\#}_{\mathrm{SA}[p]}$ is S^k-specific. We therefore update $minlen[k] = \min(minlen[k], \ell+1)$ and $minpos$ accordingly if necessary.
2. We have a non-singleton lcp interval $\ell - [lb..rb]$, we proceed by checking whether the string w of $[lb..rb]$ is string-specific. We distinguish:
   (a) If $df(w) = 1$, we therefore update $minlen[k] = \min(minlen[k], \ell+1)$ and $minpos$ accordingly if necessary. In this case is not necessary to inspect the child intervals of $[lb..rb]$ since it is obvious that they cannot change the current result for S^k.
   (b) if $df(w) > 1$ we generate the children intervals of $[lb..rb]$ and add them to the queue.

The running time of the listed algorithm is proportional to the number of visited intervals, which is $O(n)$.

Traversals Through Suffix Links

Suffix links (SLs) are a concept similar to failure links in tries. We call a triple (ψ, ϕ, c) a *suffix link* if $\overline{\psi} = cu$, where $c \in \Sigma$, $u \in \Sigma^*$ and $\overline{\phi} = u$. Whenever we are matching a string against the ST/ESA and encounter a mismatch at a vertex v, we can follow its suffix link $\phi(v)$ to effectively continue the matching. The suffix links can be computed.

Computation

SLs can be computed in a /similar fashion as the failure links in the suffix array. However, it is not so straightforward to show that the computation can be done in $O(n)$ time, since our tree is compacted.

Matching Statistics

We can utilize the suffix links to effectively compute the matching statistics of a pattern P against a string S. The matching statistics is the set

$$\mathrm{MS} = \{(i, p, \ell) \mid S[p..p+\ell] \text{ is the longest occurrence of } P_i \text{ in } S\}.$$

Chapter 6 - Making ESA Components Smaller

Suffix Array

The $\phi : [1..n] \to [1..n]$ function, defined as $\phi(i) = \mathrm{ISA}[\mathrm{SA}[i]+1]$, is increasing on each interval $[C[c]+1, C[c+1]]$ for $c \in \Sigma$. We can use some differential encoding for this, i.e. effectively encode the increasing the intervals by storing $\phi(i+1) - \phi(i)$. There is literature on how to do this effectively, while still supporting $O(1)$ random access to ϕ.

When we have ϕ in the compressed form with fast random access, we can support pattern matching in $O(m \log n)$ time by binary searching the pattern and at each position extracting the length m prefix of S – $S[\text{SA}[i]..\text{SA}[i] + m - 1]$.

Sampled Suffix Array

The compressed ϕ function serves well to answer existence resp. count queries, but not the locate queries. To find those we need to access the SA in the interval found using the binary search on ϕ. However, storing the whole SA takes a lot of space. It is possible to shrink it down by storing only the SA entries at each k-th position in text. We will also store a bitvector $B[1..n]$ in which $B[i] = 1$ iff $SA[i]$ was sampled (or $i \equiv 0 (\text{mod } k)$). This data structure is called a sampled suffix array (SSA).

We can utilize it to find the occurrence(s) of a substring in an interval $[i..j]$ by accessing $\text{SA}[o]$ for $o \in [i..j]$ by $\text{SA}[\phi^d[o]] - d$, where $o \leq k$ is the 'distance' of $SA[i]$ from the nearest sampled SA entry (a special case is $\text{SA}[\phi^d[o]] - d] < 0$). The running time of accessing $\text{SA}[i]$ is therefore $O(k)$ for a sampling rate k.

Compressed PLCP Array

Since $\text{PLCP}[i] \geq \text{PLCP}[i-1] - 1$, if we add i to both sides. Then the difference of the two subsequent PLCP values is:

$$\delta(i) = (\text{PLCP}[i] + i) - (\text{PLCP}[i-1] + i - 1) = \text{PLCP}[i] - \text{PLCP}[i-1] + 1 \geq 0,$$

which we can encode by $B = 0^{\delta(1)}1 \ldots 0^{\delta(n)}1$. This representation has n ones and at most $n - 1$ zeroes, since $\text{PLCP}[i] \leq n - 1$. In total, there is at most $2n - 1$ bits. If we preprocess the bitvector B for efficient rank/select queries, we can easily compute $\text{PLCP}[i]$ as $\text{PLCP}[i] = \text{select}(i) - 2i$, because

$$\sum_{j=1}^{i} \text{PLCP}[j] - \text{PLCP}[j-1] + 1 = \text{PLCP}[i] + i = \text{select}(i) - i$$

as the number of ones leading up to the position of i-th one is i.

We can therefore support random access to $\text{PLCP}[i]$ using only $2n - 1 + o(n)$ bits. However, what we need in many application is not access to the LCP in text order, but in the suffix array order. In that case what we would need to do for such i is find a $j = SA[i]$ and return $\text{PLCP}[j]$. This takes $O(s)$ time when we have a SSA with the sample rate of s, but as we can see in the following, there is a representation that allows $O(1)$ access to $\text{LCP}[i]$.

BPS Construction for LCP

We will first define some cool things. A *balanced parenthesis sequence (BPS)* can be formally defined as a grammar:

1. The empty string ε is a BPS.
2. If e is a BPS, then (e) is a BPS.
3. If a and b are BPSs, then ab is a BPS.

So BPSs are strings of parenthesis that correspond to valid arithmetic terms, given that some operations and variables are inserted. For example (())() is a valid BPS while (()(is not.

The second concept is a *super-cartesian tree (SCT)*, which is a cartesian tree that is allowed to have right siblings, i.e. vertices that specifically represent the same value. More formally, a SCT of an array $A[l..r]$ is constructed by:

1. Let $k = \mathrm{RMQ}(l, r)$ be the root of the current subtree T.
2. Let $\mathrm{SCT}(A[l..r])$ be the left child of T if $l < r$ (otherwise there is no left child).
3. If there is $m > k$ such that $A[k] = A[m]$, then $\mathrm{SCT}(A[k+1, r])$ is the right *sibling* of T. Otherwise $\mathrm{SCT}(A[k+1, r])$ is the right *child* of T (except the case when $r \leq k+1$, in which there is neither right child nor sibling).

The algorithm we just described can be obviously implemented so that it runs in $O(n)$ time, where n is the length of the array. If we have a SCT of an array A, we are now ready to construct its BPS. Apart from the parenthesis sequence we will use a bitvector B which contains an element for each closing parenthesis and tells us that if it corresponds to a right sibling or a right child. The BPS from a SCT can be constructed as follows:

1. Write a left parenthesis.
2. Write the BPS of the left child.
3. Write the right parenthesis.
4. Write the BPS of the right child/sibling. If it is the case of a sibling, write 0 to B, otherwise write 1 to B.

There is $C_n = \frac{1}{n+1}\binom{2n}{n}$ binary trees of n vertices, which can be approximated by the Stirling's formula to $C_n \approx \frac{4^n}{n^{\frac{3}{2}}\sqrt{\pi}} \approx 2^{2^n}$, so $\log C_n \approx 2n$. BPS of a tree is useful since it requires only $2n$ bits, which as we see is close to the information theoretic minimum and with additional $o(n)$ we can allow traversing the tree in the BPS representation. For the traversals, we require the following operations which are too complicated to describe here[2], but we will use them. In the following, we denote $(_i$ to be the position of the opening parenthesis corresponding to the value $\mathrm{LCP}[i]$. These operations are:

1. $\mathrm{rank}_((i)$ – a simple rank, for which we get $\mathrm{rank}_(((_i) = i$, similarly for $\mathrm{rank}_)(i)$.
2. $\mathrm{select}_(i)$ – a simple select for which $\mathrm{select}_(i) = (_i$, similarly for $\mathrm{rank}_(i)$.
3. findclose(i) – find the closing parenthesis corresponding to $(_i$.
4. findopen(i) – find the opening parenthesis corresponding to $(_i$.
5. enclose(i) – find the parenthesis pair $(_j)_j$ that most tightly encloses $(_i)_i$, i.e. the largest j such that findclose(i) < findclose(j).
6. rr_enclose(i, j) – range restricted enclose, i.e. the parentheses $(_k)_k$ that most tightly enclose the $(_j)_j$ and simultaneously findclose(i) < $(_k$. If there is no such pair of parenthesis, then the result is $\perp$.

However, we do not explicitly construct a SCT, but we directly construct the BPS of the SCT of A using the following algorithm:

[2] not in the book

```
LCP = [0..n] # LCP[0] = LCP[n] = -1
BPS = ['('], B = [1] # B[0] is a sentinel
s = stack([0])
for i in range(n+1):
    while LCP[i] <= LCP[s.top()]:
        s.pop()
        BPS.append(')')
        if LCP[i] == LCP[s.top()]:
            B.append(0)
        else: # LCP[i] < LCP[s.top()]
            B.append(1)
    BPS.append('(')
    s.push(i)
BPS.append('))')
B.append(01) # for LCP[0] = LCP[n] = -1
```

We can support the following operations on the BPS. We write $)_i^0$ resp. $)_i^1$ if the corresponding entry in the bitvector B is 0 resp. 1.

getParent(i, j). Given $(_i$, representing a lcp interval by the pair $(_i,)_i$, we need to find the pair of parentheses $(_j)_j$ corresponding to its parent interval. As described before, we can obtain the parent interval of

$$\text{NSV}(i) =$$

getKthChild(i, j). We have a lcp interval $\ell - [i..j]$ and want to return its k-th child. We know from before that we only need to find the ℓ-indices, from which we can easily generate the children intervals.

To do that, we start with computing $ipos = \text{select}_(i)$ and $cipos = \text{findclose}(ipos)$. Now let us look at what happens when on the stack when we write the $(_{j+1}$. Let $i_1 \leq \ldots \leq i_m$ be the ℓ-indices of the interval $[i..j]$. The stack contains $)_{i_m} \ldots)_{i_1}(_{j+1}$.

Chapter 7 - BWT

When working with large text corpora, one usually faces the problem of the scarcity of storage space as well as the need for quickly searching for relevant information. For a long time these two requirements have been perceived as conflicting in the sense that one cannot use text compression and still support efficient search queries and conversely, an indexed text cannot be significantly smaller than the plain one. To the surprise of many, both of these demands can be met simultaneously with the so-called BWT [4]. In the following text, we will first define the BWT and outline some of its properties, then in subsections and we will describe how it facilitates compression and how it can be used for text indexing, respectively. Subsection **??** briefly lists a selection of impactful applications built around the mentioned ideas.

The BWT (BWT) of a null-terminated string T of length n is a permutation of T formed by the characters that precede the suffixes of T sorted in the lexicographic order, while we treat T as circular. It was originally introduced by Burrows and Wheeler in 1994 [4] for the purpose of

text compression. Formally, let us denote M to be the matrix the rows of which consist of all lexicographically sorted rotations of T, or more specifically, $M = \left(m_{i,j}\right)_{1 \leq i,j \leq n}$, where $m_{i,1} \dots m_{i,n} = T[SA[i]..n]T[1..SA[i]-1]$. Moreover, let $F = m_{1,1} \dots m_{n,1}$ resp. $L = m_{1,n} \dots m_{n,n}$ be the string consisting of first resp. last column of M. Then we call L the BWT of T and denote it $BWT(T)$. Using as example the text $T =$ `TATGTTTTCGATG$`, we have $BWT(T) = L =$ `GGTTTCT$TAATTG` and $F =$ `$AACGGGTTTTTTT`. As one can observe, F can be obtained by simply sorting the characters of T. Fig. depicts the complete matrix M.

```
F              L
A TATGTTTTCGAT G
A TG$TATGTTTTC G
A TGTTTTCGATG$ T
C GATG$TATGTTT T
G $TATGTTTTCGA T
G ATG$TATGTTTT C
G TTTTCGATG$TA T
T ATGTTTTCGATG $
T CGATG$TATGTT T
T G$TATGTTTTCG A
T GTTTTCGATG$T A
T TCGATG$TATGT T
T TTCGATG$TATG T
T TTTCGATG$TAT G
```

Figure 6: An example of a matrix M from the BWT example in Section . Inspired by Bentley et al. [2].

Interestingly enough, the BWT is also invertible, i.e. we can reconstruct T from just $BWT(T)$. The reconstruction is done by means of the following observations. Firstly, it holds that the k-th occurrence of a character c in L corresponds to the k-th occurrence the same character in F. This is true because from the inequality $cx \prec cy$ it must hold that $x \prec y$ and so trivially also $xc \prec yc$. Therefore, for any given suffix $T[i \dots n]$ corresponding to the position j in L resp. M (i.e. $SA^{-1}[i+1] = j$) and $L[j]$ is the k-th occurrence of the character in $L[1 \dots j]$, we can find the position in L corresponding to the suffix $T[i-1 \dots n]$ by locating the k-th occurrence of $L[j]$ in F. Since the k-th row of M that ends on $L[j]$ is essentially $T[i \dots n]$ cyclically shifted rightwards for one character, we now know the corresponding position in L for $T[i-1 \dots n]$. These observations can be summarized in a bijective mapping between the columns L and F, hence it is known by the name LF-mapping and defined as

$$\mathrm{LF}(i) = \sum_{1 \leq j \leq n} [L[j] \prec L[i]] + \sum_{1 \leq j \leq i} [L[i] = L[j]], \tag{1}$$

Construction by Induced Sorting

We can construct the BWT from SA by $BWT[i] = SA[(i-1) \bmod n]$, but that requires storing the SA. However, SACAs are not very space-efficient and require at least $n \log n + n \log \sigma$ bits of memory (SA + S), which is not feasible for large memories. For this reason this is why it is more advantageous to use direct BWT construction algorithms, especially for large files/alphabets. In fact, IS can be adapted to compute the BWT directly instead of the SA.

Compression

In practice, large texts are often redundant and so the occurrences of a substring are oftentimes being preceded by only a few distinct characters. If we, for example, take the string *ssion*, there is only a few characters (*e*, *i*, etc.) that could precede it to form a valid English word. Combining this with the fact that the characters in L precede lexicographically sorted – and therefore similar – suffixes, we observe intervals in which only a few characters occur. This dramatically decreases the higher-order entropy of L and leads to the creation of long runs – maximal unary substrings. The presence of long runs enables the BWT to be much more compressible in practice, since any run $a \dots a$ of length x can be simply encoded as xa, which is also the basis of the *run-length encoding (RLE)*. We note that RLE is usually only the second stage of encoding done to a BWT and is often preceeded by a move-to-front transformation (MTF) [10, 3] as the first stage and succeeded by an efficient prefix-free code, such as Huffman code [4] as the third stage. The described procedure is essentially the basis of the popular `bzip2` [11] compression algorithm.

Backward Search

Although BWT was originally developed for the purpose of data compression, it was extended into a fully functional text index data structure in 2000 by Ferragina and Manzini [5] under the the name of FM-index. FM-index of a string T consists of a compressed representation of $BWT(T)$ enriched with several auxiliary data structures that enable efficient search queries within the compressed space.

In more detail, the need for the auxiliary data stems from the goal of supporting quick enough computation of the LF mapping. Considering that the alphabet size σ is small with respect to n, e.g. constant-sized as was assumed by Ferragina and Manzini [5], the C array of cumulative character frequencies is also small and hence can be stored as is. The efficiency of the LF mapping is therefore determined by the way we support the `rank` query. Ferragina and Manzini proposed a $O(1)$ time solution of this problem using $O((n/(\log n)) \log \log n)$ bits of additional space in terms of a RAM model with the word size of $w = \Theta(\log n)$ [5], but also other efficient solutions are known.

Ferragina and Manzini describe a *backward search* procedure, which allows to efficiently search for a pattern within the FM-index. We will now summarize the method with regards to the two types of queries supported: *count* and *locate*.

Count Queries

As before, let $P = P[1] \dots P[m]$ be the pattern of which we want to count the number of occurrences in text T. We will use the observation that the entries of the matrix M containing a suffix $P[i \dots m]$ as a prefix correspond to: *a*) exactly the occurrences of $P[i \dots m]$ in T; *b*) an interval $[s, e] \subseteq [1, n]$. We will therefore maintain the invariant that for each i we have the interval $[s_i, e_i]$, where s_i resp. e_i points to the lexicographically first resp. last row prefixed by $P[i \dots m]$. We start with $[s_{m+1}, e_{m+1}] = [1, n]$ representing the occurrences of prefix $P[m+1 \dots m] = \varepsilon$, and for each $i = m, \dots, 1$, we obtain the interval $[s_i, e_i]$ of suffixes of T which contain $P[i \dots m]$ as a prefix as $[C[P[i]] + \texttt{rank}(L, P[i], s_{i+1}), C[P[i]] + \texttt{rank}(L, P[i], e_{i+1})]$. Upon finishing the iterations, we end up with the interval $[s_1, e_1]$ of suffixes of T which contain P as a prefix, i.e. precisely the occurrences of P in T. The count query takes only $O(m)$ time and requires $|L| + O((n/(\log n)) \log \log n)$ of bits of space [5].

Locate Queries

In order to find the locations of the occurrences of P in T, we can first find the interval $[s_1, e_1]$ by the backwards search as in the counting query. However, this interval does not inherently contain the information about the actual positions of the occurrences of P. A valid option would be to use the interval $[s_1, e_1]$ to query the suffix array of T, however, storing the SA would require additional space of $O(n \log n)$. Ferragina and Manzini [5] proposed storing the SA samples at regular intervals of T of length $k \geq 0$ such that $k = \Theta(\log n)$. To locate an occurrence of P corresponding to $SA[i]$, the LF mapping is used to find the nearest stored $SA[j]$ entry to which the offset (the number of LF used mappings) is added. Using suitable data structure to store the SA samples, all the occ occurrences of P in T can be listed in $O(m + occ \log^2 m)$ using $5H_k(T) + O(\frac{\log \log n}{\log})$ space. This approach was refined in the same article to running time of $O(m + \log^{\varepsilon} n)$ and $O(H_k(T) + \frac{\log \log n}{\log^{\varepsilon} n})$ space for a constant $\varepsilon > 0$.

Wavelet Trees

Matching Statistics

The calculation of MS for strings S^1, S^2 using a BWT is quite simple. We assume that we have already computed the BWT of S^1 as well as the LCP array and preprocessed it so that we can compute the parent interval of any particular lcp interval in $O(1)$ time. Then it suffices to match S^2 in a *backwards fashion* against the BWT of S^1. The pseudocode is given in Alg. 7.

```
n1, n2 = len(S1), len(S2)
MS = [-1]*n2
lb, rb = 0, n1-1
p2 = n1-1

while p2 >= 0:
    lb, rb = backwardstep(lb, rb, S2[p2])
    if lb == n1+1: # we have an empty interval
        lb, rb = parent(lb, rb)
        l = LCP[rb]
    else:
        l += 1
        MS[p2] = l
        p2 -= 1
```

Figure 7: Matching statistics calculation using the BWT.

Let us now analyze the time complexity of Alg. 7. In each iteration of the `while` loop, we either decrement `p2` and increment the current LCP value ℓ or decrease ℓ by at least 1. The total amount of times ℓ is decreased is bounded by the number of the total increase of ℓ, which in turn is bounded by n_2. Hence if we assume the BWT of S_1, the LCP array of S_1 and other auxiliary structures are already computed, the total running time of Alg. 7 is $O(n_2)$.

MEMs

```
    n1, n2 = len(S1), len(S2)
    MS = [-1]*n2
    lb, rb = 0, n1-1
    p2 = n1-1
    path = []

    while p2 >= 0:
        lb, rb = backwardstep(lb, rb, S2[p2])
        if lb == n1+1: # we have an empty interval
            lb, rb = parent(lb, rb)
            l = LCP[rb]
            # output all MEMs of length > L
            for i, j in path:
                bad_l, bad_r = j+1, i
                for k in [i, j] - [bad_l, bad_r]:
                    if
            path = []
        else:
            l += 1
            if l > L:
                path.append( (i, j) )
            MS[p2] = l
            p2 -= 1
```

Figure 8: Calculation of all MEMs of length $> L$ using the BWT.

Shortest Unique Substrings

We can calculate SUSs using a BWT by modifying the algorithm to compute the LCP values. When we fill in a LCP$[rb]$ entry, we check whether $rb-1$ was already set and compare these two values. Similarly for the value $rb+1$, if it was already set, we can compare the current value with LCP$[rb+1]$ and output the SUS. We also need to keep track of the position of the suffix S_ℓ, since that one is always unique and hence not interesting. We do that by keeping the index of the S_ℓ in the BWT, which is 0 for $\ell = 0$ and when we move to $\ell : \ell + 1$, we update it by $idx = \text{LF}(idx)$. The complete algorithm is as follows:

```
LCP = [-2]*(n+1) # -2 stands for yet unassigned value
for c in C:
    Q.push( (C[c]+1, C[c+1]) )
size = len(C)
l = 0
idx = 0
while not Q.empty():
    lb, rb = Q.pop()
    if LCP[rb] == -2:
        LCP[rb] = l
        if LCP[rb-1] != -2 and SA[rb-1] != idx:
            m = max(LCP[rb-1], LCP[rb]) + 1
            output(SA[rb-1], m)
```

```
        if LCP[rb+1] != -2 SA[rb-1] != idx:
            m = max(LCP[rb], LCP[rb]+1) + 1
            output(SA[rb], m)
        for (i, j) in get_intervals(BWT, lb, rb):
            Q.push(i, j)
    idx += 1
    size -= 1
    idx = LF(idx)
    if size == 0:
        size = Q.size()
        l += 1
```

In fact, if we are not interested in the LCP values at all, we may simply use a bitvector B, in which $B[i] = 1$ if and only if $\text{LCP}[i] \neq -2$. We can do this, since we are filling the LCP array in increasing order from $\ell = 0$ and when we fill the $\text{LCP}[i]$ entry, we know that other filled entries have the same or less value. The time complexity remains the same as in the LCP values computation as we do only constant time overhead in each iteration in the while cycle.

Chapter 8 - Alignment

Pairwise Alignment

For two sequences S_1, S_2 of lengths n_1, n_2 over an alphabet Σ, we define the *global alignment* of S_1 and S_2 a matrix $A_{2\times m} = (a_{i,j})$ of symbols over $\Sigma \cup \{-\}$, where the '$-$' symbol is called the *gap*, in which:

1. For each $1 \leq i \leq 2, 1 \leq j, \leq m$ it holds that $a_{i,j} \in \Sigma \cup \{-\}$.

2. After deleting the $-$ symbols from $a_{i,1}, \ldots, a_{i,m}$, we get S_i for $i = 1, 2$.

3. No column in A consists of two $-$ symbols.

For a distance function δ :

$$\mathbf{NW}[i,j] = \max \begin{cases} \mathbf{NW}[i-1, j-1] + s(x_i, y_j), & \text{if } x_i \text{ is aligned to } y_j \\ \mathbf{NW}[i, j-1] - 1, & \text{if } x_i \text{ is aligned to a gap} \\ \mathbf{NW}[i-1, j] - 1, & \text{if } y_j \text{ is aligned to a gap.} \end{cases} \tag{2}$$

Hirschberg Method

Affine Gaps

Multiple Sequence Alignment (MSA)

Pruning the Search Space

2-APX Method

Heuristical Methods

Chapter 10 - Phylogenetics

We say that a semimetric is:

1. **additive**, if it satisfies $d(u,v) + d(x,z) \leq \max\{d(u,z) + d(v,x), d(u,x) + d(v,z)\}$
2. **ultrametric**, if it satisfies $d(x,y) \leq \max\{d(x,i), d(y,i)\}$ for all x, y, i

Ultrametric Trees

Additive Trees

Bibliography

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of discrete algorithms*, 2(1):53–86, 2004.

[2] Jason Bentley, Daniel Gibney, and Sharma V Thankachan. On the complexity of BWT-runs minimization via alphabet reordering. *arXiv preprint arXiv:1911.03035*, 2019.

[3] Jon Louis Bentley, Daniel D Sleator, Robert E Tarjan, and Victor K Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, 1986.

[4] Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. In *Digital SRC Research Report*. Citeseer, 1994.

[5] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398, 2000.

[6] Johannes Fischer and Florian Kurpicz. Dismantling divsufsort. *arXiv preprint arXiv:1710.01896*, 2017.

[7] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *International colloquium on automata, languages, and programming*, pages 943–955. Springer, 2003.

[8] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[9] National Center for Biotechnology Information. Overview of Structural Variation. https://www.ncbi.nlm.nih.gov/dbvar/content/overview/, Accessed Feb. 21, 2022 [Online].

[10] Boris Yakovlevich Ryabko. Data compression by means of a "book stack". *Problemy Peredachi Informatsii*, 16(4):16–21, 1980.

[11] Julian Seward. bzip2 and libbzip2. *avaliable at http://www. bzip. org*, 1996.

[12] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11. IEEE, 1973.