Finding tandem repeats in genomic data

Projects in Bioinformatics, 10 ECTS
Astrid Christiansen, 201404423
Supervisor: Thomas Mailund
Aarhus University

June 2022

Contents

1 Introduction

This project is about locating *tandem repeats* in genomic data. For DNA sequences, we look at sequences of the form $x \in \{A, C, G, T\}^*$, representing nucleotides in a DNA sequence where the letters represent the four bases, adenine (A), cytosine (C), guanine (G), and thymine (T). A tandem repeat is a sequence of one or more nucleotides that is repeated, and the repetitions are adjacent to each other in the DNA sequence. Here is an example of a tandem repeat of length 10:

$$ATCTG\ \ ATCTG.$$

Tandem repeats have a variety of uses, such as determining inherited traits in people and determining parentage. They are therefore used as an analyzing tool in certain genealogical DNA tests. The problem of finding tandem repeats in a string can be described as taking the string as input and returning the starting index and length of each tandem repeat in the string.

One algorithm that does this is described in [2]. This algorithm uses a data structure called a *suffix tree*, and can locate tandem repeats in a string x of length n in time $O(n \log n + z)$ where z is the number of tandem repeats in x. In this project, I will implement a similar algorithm, that has the same running time, but does not explicitly use a suffix tree, but instead uses a related data structure called a *suffix array* and the corresponding *LCP array*. It can be argued that this version of the algorithm is simpler, in the sense that it uses more primitive and space-efficient data structures. In this report, I will first explain how the original suffix tree algorithm works. I will then explain how suffix trees and suffix arrays are related, and how the algorithm can be modified to use a suffix array instead of a suffix tree. I will also implement the modified algorithm, and show part of my implementation. Lastly, I will show some running time experiments that I have conducted using my implemented algorithm.

2 Notation

Let $x \in \Sigma^*$ be a string over some constant size alphabet, Σ. This string, x, denotes the input string, for which we want to find tandem repeats. The string could for example be a genomic sequence, so, e.g., for DNA, we would have $x \in \{A, C, G, T\}^*$, but the algorithms are not restricted to this type of input. In practise, I choose to append a *sentinel*, \$, to x; I then let n denote the length of $x\$$. In general, a sentinel is a character that is not in Σ. I will use the character, \$, in this project; a character not in Σ, that also will be smaller than all characters in Σ. The sentinel makes it possible to create a suffix tree for x, which I will comment further on, when I have defined suffix trees.

In this project, both indices into arrays as well as indices into the string, x, are used. To avoid confusion, I will use a prime to denote indices into x. Using this notation, i would be an index into an array, while i' would be an index into x. There is not necessarily a relationship between i and i', unless this is stated.

I will use parentheses to indicate open intervals, and square brackets to indicate closed intervals. So the interval (i, j) is open on both the left and on the right, while $[i, j]$ is closed on both the left and the right. Similarly, the interval $[i, j)$ is closed on the left and open on the right, i.e., it consists of the integers $k : i \leq k < j$. I will also use intervals to denote sub-strings, e.g., $x[i, j)$ is the sub-string of x from index i to (but excluding) index j.

I denote the *depth* of a node, v, in a suffix tree as $D(v)$. By depth, I refer to the *path depth*, meaning the summed lengths of the strings on the branches leading from the root to v.

3 Finding tandem repeats using suffix trees

3.1 Suffix trees

As mentioned in the introduction, a data structure called a *suffix tree* can be used to find tandem repeats in a given string, x. A suffix tree is a compressed trie of all the suffixes of $x\$$, where \$ is the sentinel, as explained in section 2. An example of a suffix tree can be seen in Figure 1.

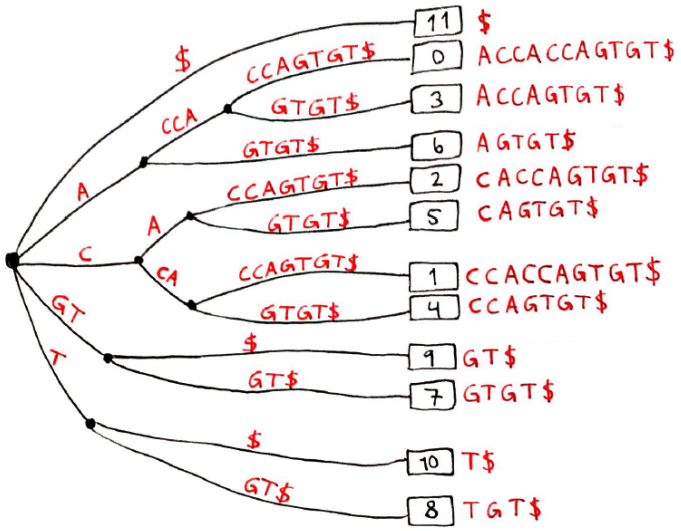

Figure 1: Suffix tree for string $x = ACCACCAGTGT\$$.

The suffix tree for x contains one leaf for each of the n suffixes of $x\$$. As mentioned in section 2, a sentinel makes it possible to create a suffix tree for x. This is because the sentinel ensures that no suffix of x is a prefix of another suffix of x. Without the sentinel, we might have a suffix which is also a prefix of another suffix, resulting in the first suffix not being a leaf in the tree, but an inner node or edge instead. But when we append the sentinel to x, this will not happen. We can construct suffix trees in time $O(n)$, e.g., using McCreight's algorithm which is described in [3]. In section 3.4, I will explain how to locate tandem repeats in a string, using the suffix tree for the string, which is the method used in [2]. Their algorithm is based on the fact that, given the suffix tree for x, what is called the *branching tandem repeats* of x can be found efficiently. Once we have these, the remaining tandem repeats can be located in a simple way.

3.2 Tandem repeats and branching tandem repeats

I have previously explained what a tandem repeat is; here follows a more formal definition that will help explain the theory further. We define a *tandem repeat* in x as a sub-string of x that consists of a repetition of a string α of length $\ell/2$, i.e., $x[i', i' + \ell) = \alpha\alpha$. As such, we can represent a tandem repeat as a pair (i', ℓ) [1]. In Figure 2, all the tandem repeats in an example string are shown.

We call a tandem repeat *branching* if the first character of the repeat is not the same as the character immediately after the repeat in x, i.e., when $x[i'] \neq x[i' + \ell]$. See Figure 3 for a visualization of the difference between a *non-branching* and a *branching* tandem repeat.

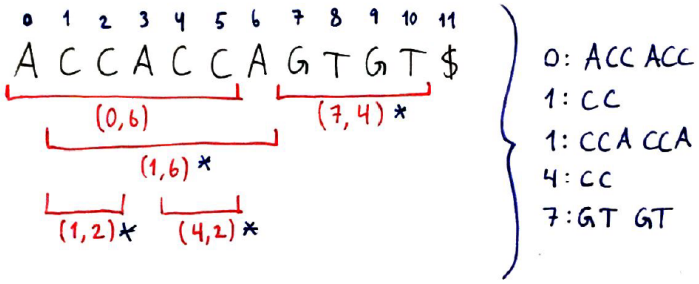

Figure 2: All tandem repeats in the string $x = ACCACCAGTGT\$$. The branching repeats are marked with an asterisk.

The algorithm in [2] starts by locating the *branching* tandem repeats, and from these, it can easily locate the rest of the tandem repeats. I will first explain how to obtain all tandem repeats from the branching ones; then I will explain how to find the branching tandem repeats.

3.3 Using branching tandem repeats to find all tandem repeats

Let the left-rotation of a string denote the modified string where the last symbol is removed and placed first in the string instead. I.e., if w is a string and a is a symbol, the left-rotation of the string wa is aw.

To obtain all tandem repeats from the branching repeats, we will use the property that every non-branching tandem repeat is a left-rotation of another tandem repeat that starts one place to its right, which may or may not be branching [2]. To be convinced of this, consider the following. By definition, a non-branching tandem repeat is a repeat (i', ℓ) where $x[i'] = x[i' + \ell]$. Let us say that $x[i', i' + \ell) = \alpha\alpha$ and $\alpha = a\beta$, where a is the first symbol of α and β is the remaining string. Since (i', ℓ) is non-branching, the symbol immediately after the repeat will also be a. So if we include the symbol after the repeat, we get $x[i', i' + \ell] = \alpha\alpha a = a\beta a\beta a$. So we see that (i', ℓ) where $x[i', i' + \ell) = a\beta a\beta$ is a left-rotation of a tandem repeat starting one place to its right, $(i' + 1, \ell)$, where $x[i' + 1, i' + 1 + \ell) = \beta a\beta a$. Therefore, every non-branching tandem repeat is a left-rotation of another tandem repeat, starting one place to its right. Also note, that this is *not* the case for *branching* tandem repeats, since $x[i'] \neq x[i' + \ell]$. This means that every non-branching tandem repeat is part of a "chain" of non-branching tandem repeats that we can obtain by rotating, and which ends with a branching tandem repeat at the right end of the chain.

This property means that all tandem repeats in a string can be found by repeated left-rotations starting from every branching tandem repeat. Therefore, since we are interested in finding all tandem repeats, we can first find all branching tandem repeats, and then continuously left-rotate these, until we have found all tandem repeats. We only have to do one character comparison, $x[i' - 1] == x[i' + \ell - 1]$, for every left-rotation to know whether we have found a tandem repeat. This means that we can find all tandem repeats in time $O(z)$, where z is the number of tandem repeats, given that we have already located the branching tandem repeats. In section 3.4, I will explain how the branching tandem repeats can be located.

3.4 Finding branching tandem repeats

If (i', ℓ) is a tandem repeat, then the suffixes i' and $i' + \ell/2$ share a prefix of length at least $\ell/2$. More precisely, they share a prefix of length exactly $\ell/2$ if the tandem repeat is branching, and more than $\ell/2$ if the tandem repeat is non-branching. See Figure 3 for a visualisation of this.

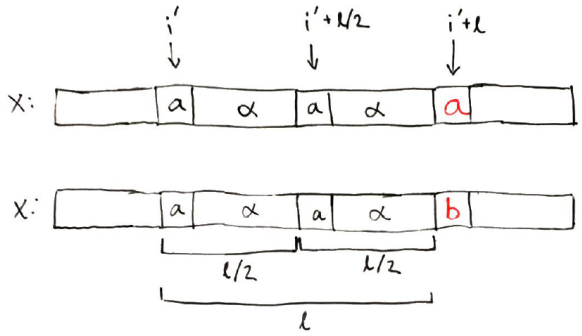

Figure 3: A tandem repeat (i', ℓ) in string x; Firstly, as a non-branching tandem repeat (top) and secondly, as a branching tandem repeat (bottom).

In Figure 3, if the repeat is non-branching (top), then suffix i' and $i' + \ell/2$ share at least $a\alpha a$ which is of length $\ell/2 + 1$. However, if the repeat is branching (bottom), then suffix i' and $i' + \ell/2$ share exactly $a\alpha$ and not more.

We can use the above observation to locate branching tandem repeats as follows, assuming that we have the suffix tree, T, of x. We process every inner node, v, of T; we know that all the leaves in the sub-tree of v share a prefix of length $D(v)$, where $D(v)$ is the depth of v. We can then check for each leaf, i', whether it is a tandem repeat of length $2D(v)$ by checking if $i' + D(v)$ is also a leaf in the sub-tree of v; see Figure 4. If this is the case, then we know that $(i', 2D(v))$ is a tandem repeat. Note, that if the repeat is branching, then the two suffixes do not share a prefix longer than $D(v)$, so they must sit in different sub-trees below v (red in Figure 4). But to actually determine if the repeat is branching, we can simply use the definition of branching tandem repeats and do the character comparison $x[i'] \neq x[i' + 2D(v)]$. Thus, the described algorithm can find all branching tandem repeats, given a suffix tree. Algorithm 1 describes the algorithm outlined above in pseudocode.

Algorithm 1 BRANCHING-REPEATS(T)

```
1: for each inner node, v, in T do
2:     for each leaf, i', in sub-tree of v do
3:         if leaf i' + D(v) is also in sub-tree of v and x[i'] ≠ x[i' + 2D(v)] then
4:             report (i', 2D(v))
5:         end if
6:     end for
7: end for
```

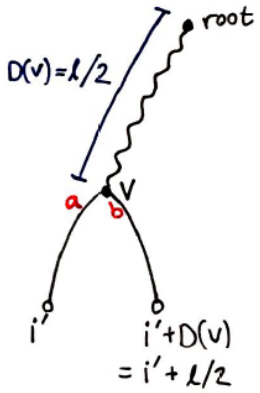

Figure 4: In the algorithm, when we process node v, we look at each leaf, i', below v, and determine whether $(i', 2D(v))$ is a tandem repeat by looking at the suffix $i' + D(v)$ in the suffix tree. If it also sits below v, then $(i', 2D(v))$ is a tandem repeat. If the repeat is branching (red), then the two suffixes sit in different sub-trees of v.

Now, we still need a way to determine if a leaf is in a given sub-tree. To do this, we preprocess T by doing a *depth-first traversal*, and assign successive numbers to the leaves as they are encountered; we then record these so-called *DFS numbers* in an array indexed by the original suffix indices. Also, for every inner node, v, we store the first and the last DFS numbers assigned in the sub-tree of v; this gives us the interval of DFS numbers that are in the sub-tree of v. With this interval stored at v, we can easily check if a leaf is in the sub-tree of v; we simply check if the DFS number of the leaf is in the DFS interval of v. Using this method, we can check if a leaf is in a given sub-tree in constant time; we just look up in an array and check if the resulting value is within the given interval. So the algorithm finds all branching tandem repeats in time proportional to the total size of all the sub-trees. The worst case input is therefore a string consisting only of one character (and the sentinel), e.g., $x = AAAAAA\$$; see Figure 5.

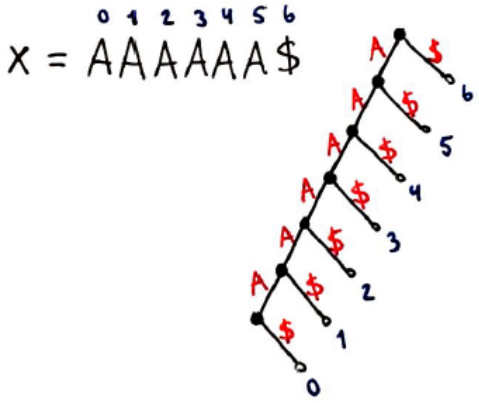

Figure 5: Suffix tree for $x = AAAAAA\$$ (a worst case scenario for finding tandem repeats).

The suffix tree for such a string has n inner nodes, and the total size of all the sub-trees is $\sum_{i=0}^{n-1} n - i = \Theta(n^2)$. However, we can use a method known as *the smaller half trick* to improve the running time to $O(n \log n)$.

3.5 The smaller half trick

In the case of Algorithm 1, the smaller half trick can be described simply as *not processing* the widest child node of each inner node, v. Here, the width refers to the DFS interval stored at each inner node, i.e., the widest node is the node with the most leaves below it. So why does this improve the running time to $O(n \log n)$? To understand this, let us consider a single leaf, i'. How many times could we potentially look at i' in the algorithm? Without using the smaller half trick, the number of times we look at i' is $O(n)$, since we have n inner nodes in the worst case. Now, if we skip the processing of the widest child node for every inner node, how many times we could we then end up looking at leaf i'? We only look at i' from the ancestors that have a wider sibling, when we use the smaller half trick. The parent of such an ancestor clearly has a width at least double that of the ancestor. So for each ancestor from which we look at i', if we move up one ancestor in the tree, the width doubles. We can only double the width $O(\log n)$ times before we reach a width of n (the root), so we can only look at i' from $O(\log n)$ ancestors. Therefore, since we have n leaves in total, the running time of the algorithm using the smaller half trick is $O(n \log n)$. Now that we understand why the smaller half trick works, let us consider how to use it in the algorithm.

When we choose to skip the processing of these widest nodes, we could end up missing some branching tandem repeats. We fix this problem in the following way. Let us assume that we have a branching tandem repeat, $(i', 2D(v))$, below some node, v. Then, when processing node v, suffix i' and $i' + D(v)$ will both be below v, but in different sub-trees of v. Therefore, if one of these suffixes is below the widest node of v, then the other will not be. If suffix $i' + D(v)$ is below the widest node of v, then suffix i' is not, so when we process leaf i', then we "look forwards" to $i' + D(v)$ which is below the widest node of v and find that i' is indeed a branching tandem repeat. However, if it is i' that is below the widest node of v and $i' + D(v)$ that is not, then we do not process i', but only $i' + D(v)$. So in order to locate i' as a branching tandem repeat, we have to also "look backwards" from $i' + D(v)$ by subtracting $D(v)$. Then, we find i' below the widest node of v, and find that it is a branching tandem repeat. Thus, using the smaller half trick, we can avoid processing the widest child node of each inner node, without missing any branching tandem repeats. The modified algorithm is described as pseudocode in Algorithm 2.

Algorithm 2 BRANCHING-REPEATS-SMALLER-HALF(T)

```
1: for each inner node, v, in T do
2:     for each leaf, i', in "sub-tree of v except widest(v)" do
3:         if leaf i' + D(v) is also in sub-tree of v and x[i'] ≠ x[i' + 2D(v)] then
4:             report (i', 2D(v))
5:         end if
6:         j' ← i' − D(v)
7:         if leaf j' is also in widest(v) and x[j'] ≠ x[j' + 2D(v)] then
8:             report (j', 2D(v))
9:         end if
10:    end for
11: end for
```

So we can use Algorithm 2 to find all *branching* tandem repeats in time $O(n \log n)$, and then use the rotating method described in section 3.3 to find the remaining tandem repeats in time $O(z)$, where z is the number of tandem repeats in x. So now, we have all the tools needed for finding all tandem repeats in x in time $O(n \log n + z)$, using the suffix tree for x. In section 4, I will explore how to modify the suffix tree algorithm to use a different data structure called a *suffix array* instead.

4 Finding tandem repeats using suffix arrays

In section 3, I explained how to find all tandem repeats in a given string using the suffix tree for the string. In this section, I will explain how the same algorithm structure can be used for locating tandem repeats using a *suffix array* instead of a suffix tree.

4.1 The relationship between suffix trees and suffix arrays

A suffix array, SA, of a string, x, is an array containing the suffixes of x in lexicographical order. In practise, the suffix array typically contains the suffix indices into x and not the actual suffixes. See Figure 6 for an example of a suffix array.

```
Suffixes of x and their indices:
0:  ACCACCAGTGT$
1:  CCACCAGTGT$
2:  CACCAGTGT$
3:  ACCAGTGT$
4:  CCAGTGT$
5:  CAGTGT$
6:  AGTGT$
7:  GTGT$
8:  TGT$
9:  GT$
10: T$
11: $
```

```
Suffix array, SA, of x:
 SA[0] = 11 ($)
 SA[1] = 0  (ACCACCAGTGT$)
 SA[2] = 3  (ACCAGTGT$)
 SA[3] = 6  (AGTGT$)
 SA[4] = 2  (CACCAGTGT$)
 SA[5] = 5  (CAGTGT$)
 SA[6] = 1  (CCACCAGTGT$)
 SA[7] = 4  (CCAGTGT$)
 SA[8] = 9  (GT$)
 SA[9] = 7  (GTGT$)
SA[10] = 10 (T$)
SA[11] = 8  (TGT$)
```

Figure 6: Suffixes and suffix array for string $x = ACCACCAGTGT\$$. The suffix array contains the indices of the sorted suffixes of x.

We could construct a suffix array from a suffix tree by doing a depth-first traversal of the tree in lexicographical order, and recording the leaves in the order that we encounter them. This is similar to what we did in section 3.4, but there we did not necessarily traverse the tree in lexicographical order. However, we can do this without increasing the time complexity, since we have a constant size alphabet. Then, when we run through T in this depth-first order, we will see the leaves in the order $SA[0], SA[1], ..., SA[n-1]$; see Figure 7. In Figure 7, I have also included the inverse suffix array, ISA, of x, since we will use this later; we can easily construct ISA from SA, entry by entry, in time $O(n)$.

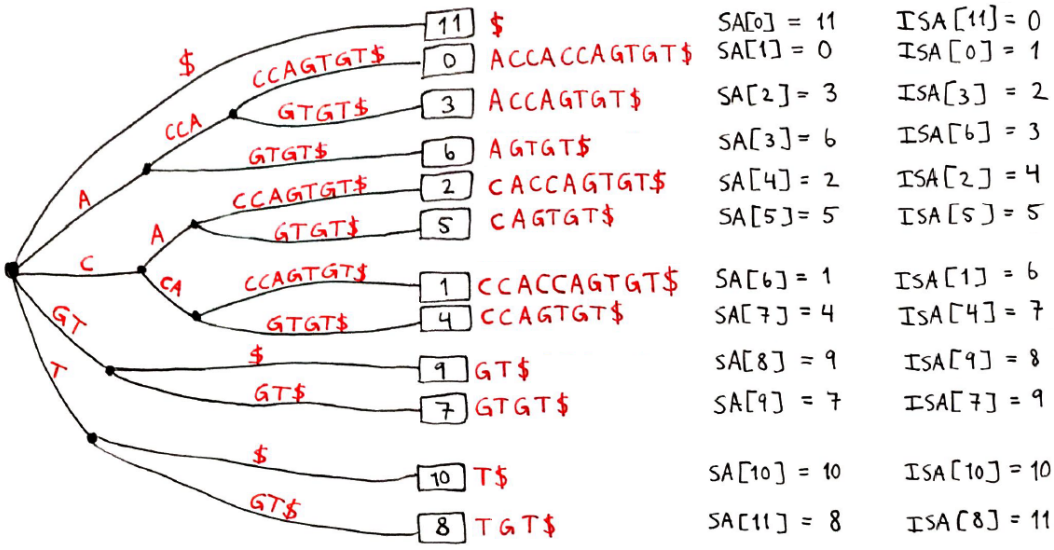

Figure 7: Relationship between suffix tree, suffix array and inverse suffix array for string $x = ACCACCAGTGT\$$.

A suffix array is not able to replace a suffix tree by itself. There is some information in T that we do not have stored in SA; namely the structure of the tree, given by the inner nodes. We will store this information in another array called the *longest common prefix array* (*LCP array*). $LCP[i]$ contains the length of the longest shared prefix of the two suffixes, $SA[i]$ and $SA[i-1]$. We choose to set $LCP[0] = 0$. See Figure 8 for an example of an LCP array.

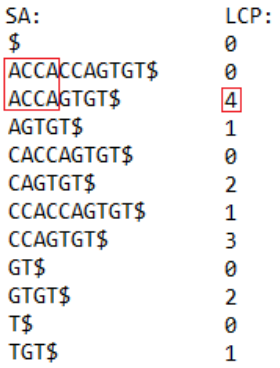

Figure 8: Suffix array (as actual suffixes) and LCP array for string $x = ACCACCAGTGT\$$. $LCP[i]$ contains the length of the longest shared prefix of $SA[i]$ and $SA[i-1]$.

If we have both the suffix array and LCP array of a string, we can create the corresponding suffix tree, since the suffix array gives us the lexicographical depth-first order of the suffix tree leaves, and the LCP array can be used to find the locations of the inner nodes in the tree. This means that with the suffix array and LCP array, we have the same information as we do with the suffix tree. However, if we wish to retain a running time of $O(n \log n + z)$ for finding tandem repeats, we cannot use more time than this on creating the two arrays. If we for example were to naïvely sort the suffixes to obtain the suffix array, this could be done in time $O(n^2)$, e.g., using radix sort, but this would be too slow. However, there are methods, such as the Skew algorithm which is described in [3], that allows us to construct the suffix array in time $O(n)$. I will not go into details about the algorithm here, as it is not the primary focus of this project, but I have implemented and used the algorithm; my implementation can be seen at https://github.com/Allegris/TR-project-F2022. The LCP array can also be constructed in time $O(n)$, using the suffix array; an algorithm for this is also described in [3], and my implementation can be seen following the link above.

Given that we can store the same information in SA and LCP as in the suffix tree, T, we should be able to modify the algorithm explained in section 3.4 to use these arrays instead of T. In the suffix tree algorithm, we process the inner nodes in T. So in order to reuse the structure of the algorithm, we need something similar to the inner nodes in T, using SA and LCP.

4.2 ℓ-intervals

As a representation of the inner nodes in T, we will use so-called *ℓ-intervals* [1]. We define an ℓ-interval as an interval $[i, j)$ into the suffix array where the suffixes $x[SA[k]]$ for $k \in (i, j)$ all share a prefix of length ℓ, but do not all share a longer prefix. Furthermore, the potential two suffixes surrounding the interval in the suffix array must not share a prefix of length ℓ or longer with the interval suffixes. In summary, we define an ℓ-interval $[i, j)$ as an interval where

$$\begin{aligned}
&\ell = min_{k \in (i,j)} LCP[k] \quad \text{and either} \\
&i = 0 \text{ or } LCP[i] < \ell \quad \text{and either} \\
&j = n \text{ or } LCP[j] < \ell
\end{aligned}$$

Since $LCP[i] < \ell$, we cannot extend the interval to the left and still have an ℓ-interval, and since $LCP[j] < \ell$, we cannot extend the interval to the right and still have an ℓ-interval. From the definition, we can see that the ℓ-intervals correspond to the nodes in the suffix tree at depth ℓ; see Figure 9. We will use LCP to locate the ℓ-intervals in SA in a recursive manner, always choosing the smallest relevant ℓ; this way, we indirectly construct the suffix tree from the root and down to the leaves.

First, we look at the entire LCP array. We then identify the smallest value, ℓ, within it (of course, ignoring index 0). Then, we split the LCP array at every occurrence of ℓ; this will result in some child intervals that correspond to the first layer of "inner nodes". We then recurse on each of the child intervals. Note, that in each recursion, we ignore the first index in the input interval, as this value indicates how much the corresponding suffix shares with the suffix above, i.e., a suffix outside of the interval, which is not relevant. We stop the recursion, when the input interval is a singleton interval, $[i, i + 1)$, i.e., when we have hit a leaf. See Figure 9 for a visualisation of the algorithm, and how the ℓ-intervals correspond to the inner nodes in the suffix tree.

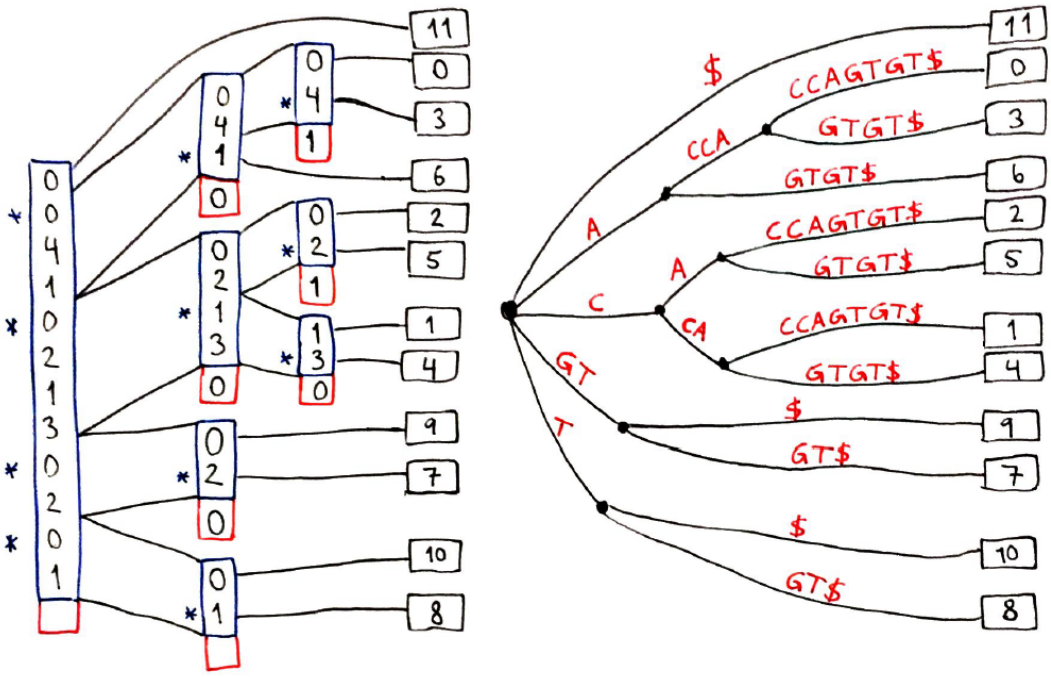

Figure 9: Relationship between ℓ-intervals and inner nodes in suffix tree for string $x = ACCACCAGTGT\$$. The dark blue boxes represent ℓ-intervals, $[i, j)$. I have included the LCP value at index j as red boxes below the ℓ-intervals; these are not included in the intervals, but we can see from them, that the constraints for ℓ-intervals are satisfied. The leaves are not represented as intervals, but are of course singleton intervals, $[i, i+1)$. I have also drawn where the occurrences of ℓ are located in each interval (as asterisks), i.e., the points where the interval is split.

Now that we have a representation of the inner nodes in T, i.e., the ℓ-intervals, we also need a way to check if a leaf is in the sub-tree of a given inner node. Given an inner node in the form of an ℓ-interval in SA, $[i, j)$, we can easily check whether a given leaf, k', is in the sub-tree of the inner node, using the inverse suffix array, ISA, of x. If $ISA[k'] \in [i, j)$, then suffix k' is a leaf below the node corresponding to the ℓ-interval $[i, j)$; see Figure 7. This way, we can identify leaves within a sub-tree in constant time using the inverse suffix array. In section 4.3, I will explain a method for constructing the ℓ-intervals, given SA and LCP.

4.3 Using RMQ to identify ℓ-intervals

Now we have outlined an algorithm for finding the ℓ-intervals in SA, which correspond to the inner nodes in the suffix tree. We simply find the occurrences of ℓ, split at these points, and recurse. But *how* do we find the occurrences of ℓ? We want to "grow the tree" from the top down, so we always want to choose the smallest ℓ that is relevant. We do this using the so-called *range minimum query* (*RMQ*) [1]. RMQ can be described as the problem of finding the index of

the first occurrence of the minimum value in a given (non-empty) interval of an array. For our purpose, for an array A, we let $A.RMQ[i, j) = (k, A[k])$ where $A[k] = \min_{q \in [i,j)} A[q]$; so we look from index i to $j - 1$ in A, and we return both the index of the first occurrence of the minimum value, and the value itself. I will refer to $A[k]$ as the *RMQ value* of $(k, A[k])$.

So how do we use RMQ to find the occurrences of ℓ in LCP, i.e., the split points in the algorithm? We start by looking at the entire LCP, and use RMQ to find the first split point (again, we ignore the first index, here $LCP[0]$). Let this first split point have index i_1 and let $\ell = LCP[i_1]$. Then we use RMQ on the rest of the array, from index $i_1 + 1$ to the end, and if the found RMQ value is still ℓ, then we have found the next split point. We continue doing this, until we have no more array left, or until we find a RMQ value larger than ℓ, meaning that we have no more split points. This described algorithm is shown as pseudocode in Algorithm 3. Then, we recurse on all the found child intervals, until we reach singleton intervals, i.e., leaves of the suffix tree.

Algorithm 3 FIND-CHILD-INTERVALS[i, j)

```
(i_prev, ℓ) = RMQ[i + 1, j)
report [i, i_prev)
(ii, ℓℓ) = RMQ[i_prev + 1, j)
while ℓℓ = ℓ do
    report [i_prev, ii)
    i_prev = ii
    (ii, ℓℓ) = RMQ[i_prev + 1, j)
end while
report [i_prev, j)
```

But how do we compute RMQ? We could preprocess the RMQ for every possible interval and store these in an $(n \times n)$ matrix, M, where $M_{ij} = A.RMQ[i, j)$. Using dynamic programming, we could fill out M in time $O(n^2)$. This way, we could answer queries in constant time. However, the preprocessing time of $O(n^2)$ is not desirable for our purpose, since we want to be able to find all tandem repeats in time $O(n \log n + z)$. There are multiple solutions to RMQ that will achieve a better running time; I will use a method with a preprocessing time of $O(n \log n)$ (and still constant query time), since this is within the desired time complexity.

4.3.1 RMQ in time $O(n \log n)$

In the faster RMQ method, we will indeed fill out a matrix, M, but this time, it will not directly contain the RMQ for every possible interval. The matrix will not be an $(n \times n)$ matrix, but an $(n \times \log n)$ matrix; we then only calculate the RMQ for intervals of lengths that are a power of 2; so 1, 2, 4, 8, 16, etc. The idea is then that we can answer any query, using only these. First, I will explain why this is the case.

If our query interval has a length that is a power of 2, we can simply look up the RMQ in M. If its length is not a power of 2, then we can still "cover" the interval with shorter intervals. We can choose to cover the interval with two intervals of the greatest possible length that is smaller than the query length, but *is* a power of 2; see Figure 10. When determining the query RMQ, we then compare the RMQ values for both of these intervals, and choose the RMQ with the minimum RMQ value as the query RMQ. The two RMQs that we want to compare are both in M. It does not affect the result, that the intervals overlap, since this will not change the minimum RMQ value.

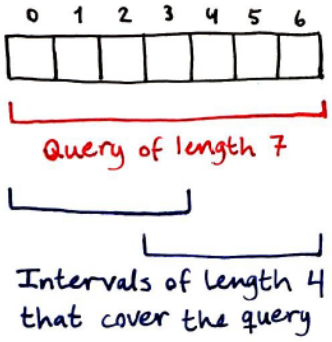

Figure 10: Answering a query of length 7 by covering it with two intervals of length 4.

Now, let us consider how we can fill out M. It is clear, that an interval of length 2^i can be split into two intervals of length 2^{i-1}. So when we fill out M, if we have already filled all the entries for intervals of length 2^{i-1}, then we just need to compare two RMQ values found by table look-ups per entry of length 2^i. So we fill out M column by column, such that we always have the RMQs for the 2^{i-1}-intervals, when we calculate the RMQs for the 2^i-intervals. Then each matrix entry takes constant time to calculate, so the total time for filling out M is $O(n \log n)$. I have implemented this version of RMQ for my algorithm for finding tandem repeats. My implementation can be seen at https://github.com/Allegris/TR-project-F2022.

4.4 Finding branching tandem repeats using suffix arrays

Now that we have a method for finding ℓ-intervals, how do we use these intervals to locate the branching tandem repeats? We will use the same structure as in section 3, but now we have arrays and ℓ-intervals instead of trees and nodes. I have previously explained that the ℓ-intervals correspond to the nodes at depth ℓ in the suffix tree. In section 3, we iterated the inner nodes; we will now iterate the ℓ-intervals instead. In order to explain the algorithm, let us assume, that we are about to process the ℓ-interval, $[i, j)$; see Figure 11.

The ℓ-interval, $[i, j)$, corresponds to an inner node, v, at depth ℓ in the suffix tree, and the sub-trees below v correspond to child intervals, $[i, i_1) \cdot [i_1, i_2) \cdot \ldots \cdot [i_m, j)$. For a child interval, $[i_k, i_{k+1})$, the "leaves" below are $SA[q]$ for $q \in [i_k, i_{k+1})$. We want to check if each such leaf is a branching tandem repeat. We do this by checking if $SA[q] + \ell$ is within the interval $[i, j)$, but in a different child interval. We can do this by checking if $SA[q] + \ell$ is in either the interval $[i, i_k)$ or $[i_{k+1}, j)$; again, see Figure 11. In the tree, this corresponds to the leaf being in a different sub-tree below v, which is the case exactly for the branching tandem repeats. So this is how we locate the branching tandem repeats in the suffix array.

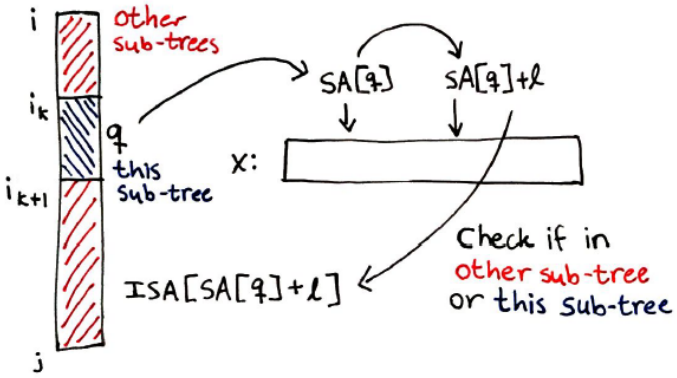

Figure 11: Locating branching tandem repeats using SA.

Assuming that we already have the child intervals, iterating the indices takes time linear to the size of the interval, since we only do constant time table look-ups for each index. However, we have to do this for every ℓ-interval that we can find by recursion, so the total time is again $\Theta(n^2)$. Therefore, it is time to use the smaller half trick, once again, to get a running time of $O(n \log n)$!

4.4.1 The smaller half trick for suffix arrays

In section 3.5, I described the smaller half trick as *not processing* the widest child node of every inner node, v. We can do the same here, only now, we avoid processing the widest child interval of every ℓ-interval. The idea is shown in Figure 12.

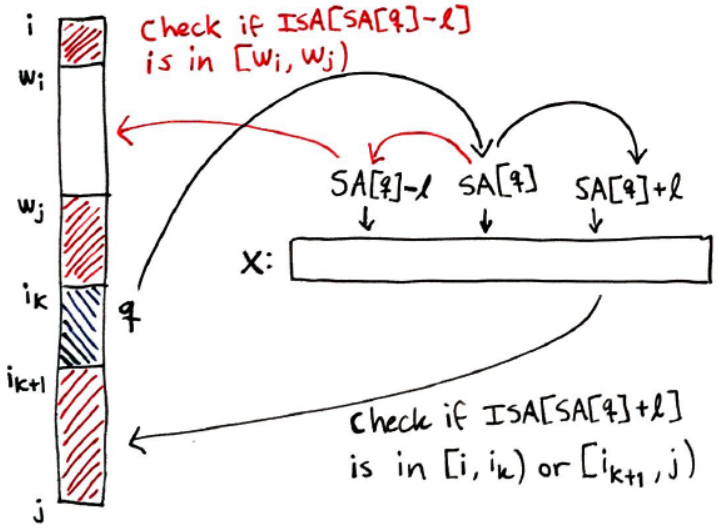

Figure 12: Locating branching tandem repeats using SA and the smaller half trick for improving the running time.

Again, doing this, we could risk missing some tandem repeats; we fix this by not just "looking forwards" to $SA[q]+\ell$, but also "looking backwards" to $SA[q]-\ell$, as shown in Figure 12. The reasoning for why the smaller half tricks works is described in section 3.5, so I will not discuss it again here. Once we have used this algorithm to find all branching tandem repeats, we can again use the rotating method described in section 3.3 to find the remaining tandem repeats. This means, that we now have everything needed for locating all z tandem repeats in a string of length n in time $O(n \log n + z)$, constructing and using the suffix- and LCP arrays for the string.

5 Comparison of the two algorithms

The suffix tree algorithm described in section 3 and the suffix array algorithm described in section 4 are very similar in structure. They also have the same time complexity, $O(n \log n + z)$. The biggest difference between them is the data structures that they use. Suffix trees and suffix arrays are closely related, but suffix arrays are more space-efficient and have a straightforward representation. Especially the smaller space consumption of suffix arrays is a crucial reason for preferring these over suffix trees [3]. I would argue that the suffix array algorithm is simpler in this

way, since it only uses arrays and do not require the relatively more complicated suffix tree data structure. For educational purposes, though, I think the suffix tree algorithm is easier to visualise and therefore understand. But for implementation, I prefer the suffix array algorithm, because of its space-efficiency and simple data structure. So I have chosen to implement the suffix array algorithm, and I will show part of my implementation in section 6.

6 Implementation

In this section, I will show part of my python implementation of the suffix array algorithm for finding tandem repeats. I will show two functions; the one for finding branching tandem repeats, using the smaller half trick, and the one for finding all the tandem repeats from the branching ones, using left-rotations. The other functions that I have implemented in order to make these work, such as the Skew algorithm, the LCP construction algorithm, RMQ, finding child intervals, etc., I will not include here, but they can all be seen at https://github.com/Allegris/TR-project-F2022.

I have implemented the algorithm for finding branching tandem repeats, using the smaller half trick, as below:

```
def branching_TR_smaller_half(x, sa, lcp):
    isa = construct_isa(sa)
    M = RMQ_preprocess(lcp)
    for (i, j) in get_inner_nodes(lcp, M, 0, len(x)):
        child_nodes = list(get_child_nodes(lcp, M, i, j))
        (w_i, w_j) = widest(child_nodes)
        (_, L) = RMQ(lcp, M, i + 1, j)
        for (ii, jj) in child_nodes:
            if (ii, jj) == (w_i, w_j):
                continue
            for q in valid_isa_index(sa, ii, jj, +L):
                r = isa[sa[q] + L]
                if (i <= r < j) and not (ii <= r < jj):
                    yield (sa[q], 2*L)
            for q in valid_isa_index(sa, ii, jj, -L):
                r = isa[sa[q] - L]
                if w_i <= r < w_j:
                    yield (sa[r], 2*L)
```

The function takes x, the suffix array and the LCP array as inputs, and finds the branching tandem repeats using the method described in section 4.4.1. In my implementation, I have chosen to name my variables after the suffix tree terms, rather than the equivalent suffix array terms, since I think this makes it easier to visualise what the code does; e.g., I use the term "inner nodes" to denote what are actually ℓ-intervals. In line 2 of the code, I construct the inverse suffix array, *isa*, from the suffix array. In line 3, I preprocess the $(n \times \log n)$ RMQ matrix, M, as described in section 4.3.1. I then find and iterate the inner nodes. I find the inner nodes using a separate, recursive function, *get_inner_nodes*(), implemented using a stack; the function corresponds to the algorithm visualised in Figure 9. For each inner node, I find the direct child nodes, using *get_child_nodes*(), which corresponds to Algorithm 3. In line 6, I call a function that identifies the widest child node. I then find the length, L, of the longest shared prefix of all the leaves below the inner node, using RMQ on the LCP array; i.e., L is the depth of the inner node. Then, I iterate the child nodes, except the widest one, and for each leaf below, q, I check left (lines 11-14) and right (lines 15-18), as described in section 4.4.1.

Once I have located all the branching tandem repeats, I use these to find all the tandem repeats by repeated left-rotations. My implementation of this is as follows:

```
1  def find_all_tandem_repeats(x, branching_TRs):
2      for (i, L) in branching_TRs:
3          yield (i, L)
4          while can_rotate(x, i, L):
5              yield (i-1, L)
6              i -= 1
7
8  def can_rotate(x, i, L):
9      return i > 0 and x[i - 1] == x[i + L - 1]
```

Above, I iterate the branching tandem repeats. For each branching repeat, I yield it, and then I check if I can rotate left, and still get a tandem repeat. I continue left-rotating, as long as possible, yielding the tandem repeats found in the process. Once I have done this for all the branching repeats, I am done, and have now located all the tandem repeats in x.

7 Experiments

In this section, I will show some experiments that I have conducted, using my implemented algorithm. First, I tested the correctness of the implementation, and then I measured the running time for different types of data.

7.1 Correctness

I have tested my implementations on multiple handmade examples, as well as some randomly generated sequences. It is easy to check if the resulting tandem repeats are in fact tandem repeats. It is more difficult to be convinced that all the tandem repeats have indeed been found. So I have run the algorithm on many short sequences, for which I were able to find the tandem repeats with confidence by hand, and compared them to the results of my implementation. The conclusion from this is, that my algorithm seems to work as expected. In Figure 13, two examples of prints of the tandem repeats found by my implementation can be seen.

```
**********
TANDEM REPEATS, (index, length), for string:
x: ACCACCAGTGT$
(0, 6): ACC ACC
(1, 2): C C
(1, 6): CCA CCA
(4, 2): C C
(7, 4): GT GT
**********
```

```
**********
TANDEM REPEATS, (index, length), for string:
x: aaaaaa$
(0, 2): a a
(0, 4): aa aa
(0, 6): aaa aaa
(1, 2): a a
(1, 4): aa aa
(2, 2): a a
(2, 4): aa aa
(3, 2): a a
(4, 2): a a
**********
```

Figure 13: Implemented algorithm output for string $x = ACCACCAGTGT\$$ and string $x = aaaaaa\$$.

When the input string is of form $A^n\$$, then the number of tandem repeats should be $z = \Theta(n^2)$, which indeed seems to be the case; see Figure 14. In conclusion, the algorithm does indeed seem to locate all the tandem repeats.

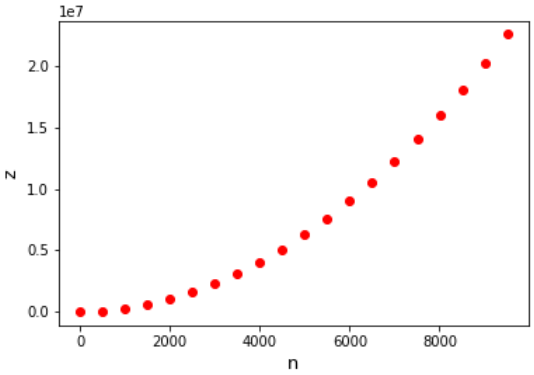

Figure 14: The number of tandem repeats found, z, as a function of n for sequences of form $A^n\$$.

7.2 Worst case running time

I have also tested the running time of my implementation. Since the total running time of the algorithm, $O(n \log n + z)$, depends both on n and z, I have measured the running time for finding branching tandem repeats (expected time $O(n \log n)$), and the algorithm for finding all tandem repeats (expected time $O(z)$), separately. First, I ran the two algorithms on the worst possible input, i.e., $A^n\$$, for n up to 10,000. For each sequence, I ran the algorithm five times, and I have used the average running times in the following plots.

The result for the branching tandem repeat algorithm can be seen in Figure 15. The left plot in Figure 15 shows the running time. The right plot shows the running time divided by the expected running time, $n \log n$. Since the right plot seems to lie beneath a constant line, as n grows, the running time does indeed seem to be $O(n \log n)$.

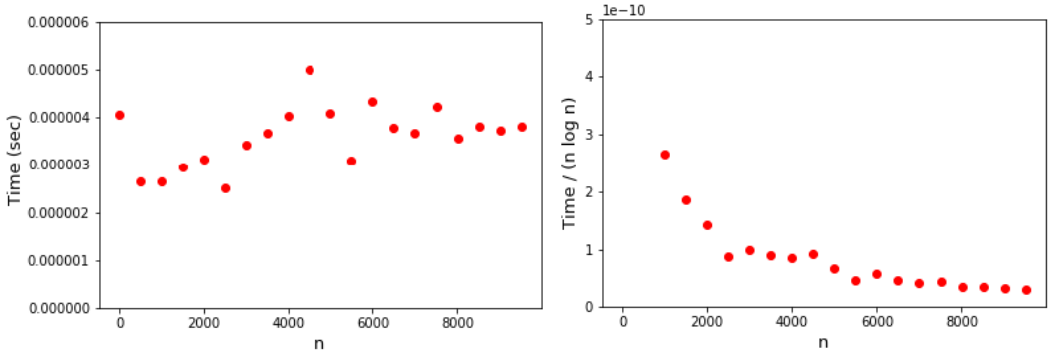

Figure 15: Left: Worst case running time for finding branching tandem repeats. Right: Worst case running time divided by expected worst case running time, $n \log n$, for finding branching tandem repeats.

Corresponding plots for the rotation algorithm for finding all tandem repeats can be seen in Figure 16. Again, the left plot shows the running time, and the right plot shows the running time divided by the expected running time, which is now z. Again, since the right plot seems to lie beneath a constant line, as n grows, the running time does indeed seem to be $O(z)$.

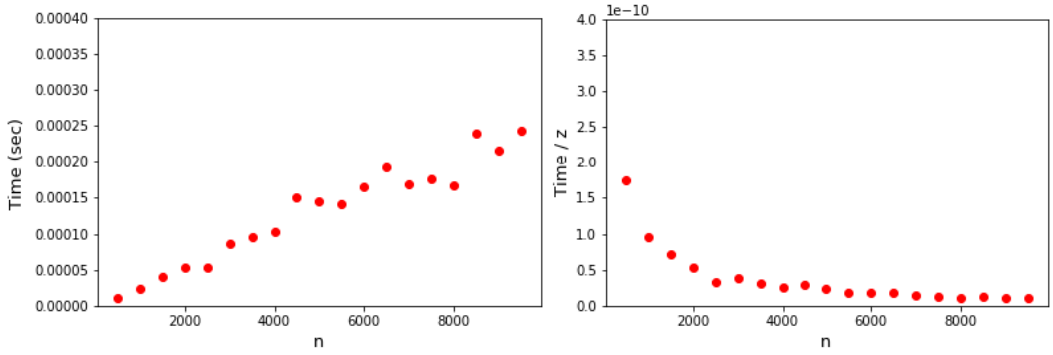

Figure 16: Left: Worst case running time for finding all tandem repeats. Right: Worst case running time divided by expected worst case running time, z, for finding all tandem repeats.

7.3 Running time for random data

I have also tested the algorithm on randomly generated sequences, $x \in \{A, C, G, T\}^n$, again for n up to $10,000$. For each sequence, I again ran each algorithm five times and plotted the average running times. The plots for the branching tandem repeat algorithm can be seen in Figure 17. As we would expect, the algorithm also seems to be $O(n \log n)$ for random sequences.

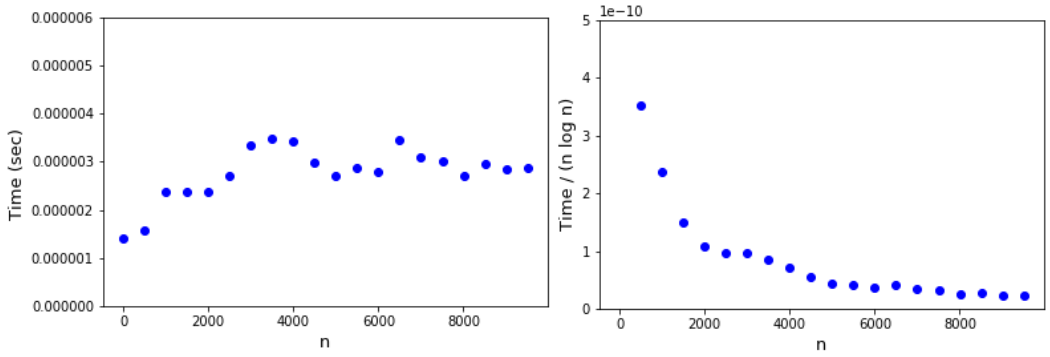

Figure 17: Left: Running time for finding branching tandem repeats in random sequences. Right: Running time for random sequences divided by the expected worst case running time, $n \log n$, for finding branching tandem repeats.

Similar plots for the algorithm for finding all tandem repeats can be seen in Figure 18. We see that the right plot is close to a horizontal line, again indicating that the running time is $O(z)$.

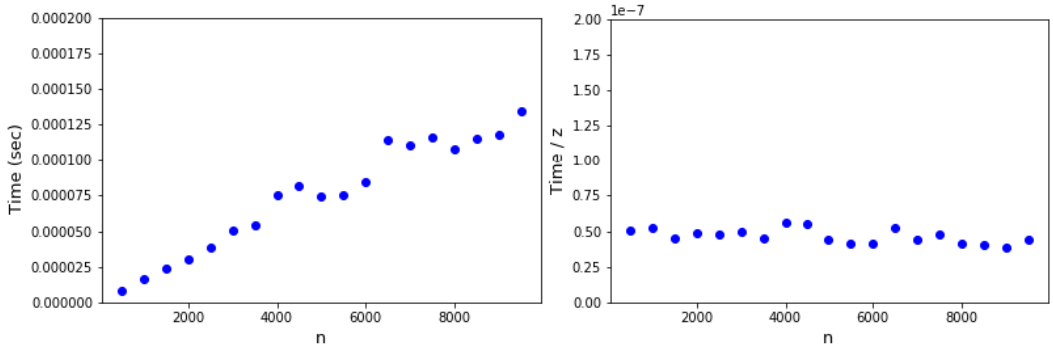

Figure 18: Left: Running time for finding all tandem repeats in random sequences. Right: Running time divided by expected running time, z, for finding all tandem repeats in random sequences.

From these experiments, it seems that the implementation works as expected, and that the algorithm does indeed have a running time of $O(n \log n + z)$, which is the same as the suffix tree algorithm.

8 Conclusion

In this project, I have examined two different ways of locating tandem repeats in strings such as DNA sequences. First, I explained how this could be done in time $O(n \log n + z)$, using suffix trees. Then, I explained how suffix trees are related to another data structure, suffix arrays, and how the algorithm could be modified to use suffix arrays instead, retaining the same time complexity. The two algorithms are very similar in structure, but use different data structures. I have compared the two algorithms, concluding that the suffix tree algorithm might be the best for understanding how and why the algorithm works, since it is easy to visualise; however, the suffix array algorithm might be better for implementation, due to its space-efficiency and simple data structures. I lastly showed part of my implementation of the suffix array algorithm, and the results of some experiments indicating that the running time of the suffix array algorithm is indeed $O(n \log n + z)$.

9 References

[1] - Computing branching repeats, note by Thomas Mailund at BiRC at Aarhus University, 2022.

[2] - Simple and flexible detection of contiguous repeats using a suffix tree, by Jens Stoye and Dan Gusfield, University of California, 2001.

[3] - String Algorithms in C, by Thomas Mailund, 2020, ISBN:978-1-4842-5920-7.