Projects in Bioinformatics

AARHUS UNIVERSITET

October 31, 2016

Implementation and practical runtime of the Sanders suffix array algorithm versus the naive implementation

Author:
Marni Tausen

Supervisor:
Christian N. S. Storm

Abstract

The aim of this project was to implement both the naive and Sanders suffix array construction algorithms, and test their practical running times on various input data. Both algorithms were implemented in C using only standard libraries. The running time experiments show that they have a worst case running time of $O(n^2)$ and $O(n)$ respectively, but the naive algorithm typically performs better in practice due to its simple construction. This shows that for most common data, like DNA sequences and regular text, you would typically get much faster run times with just using the naive implementation, where the Sanders implementation gives a much more stable runtime for any given input.

Contents

1 Introduction **1**

2 Suffix array **1**

3 Sanders suffix array construction **3**

4 Implementations **6**

5 Practical running times **7**

6 Conclusion **13**

1: Introduction

Suffix arrays and suffix trees are widely used data structures for general string processing and in bioinformatics. Applications within bioinformatics are relevant as major part of the data is strings, typically in the form of DNA or protein sequences, where we want to find particular patterns in those sequences. Both describe re-occurring patterns in strings, through summarization with all suffixes of the particular strings. Common uses for suffix arrays and suffix trees, include pattern matching, finding tandem repeats, common substrings, read mapping and etc. Essentially any string matching problem can be solved using either a suffix array or a suffix tree.

A suffix of a string is a substring, which starts at a specific i and includes the remaining part of the string. We can describe a suffix as the following function:

$$\text{suffix}_i(\text{string}) = \text{string}[i \dots n] \; \forall \; 1 \leq i \leq n \tag{1}$$

All strings have as many suffixes as characters, as we can always make a suffix starting from that character, by including the remainder of the string.

The opposite of a suffix is a prefix, where we include all of the characters up to a certain i.

$$\text{prefix}_i(\text{string}) = \text{string}[1 \dots i] \; \forall \; 1 \leq i \leq n \tag{2}$$

This report looks at the naive suffix array construction algorithm, and the Sanders and Kärkkäinens linear time suffix array construction algorithm [1]. Both of the algorithms were implemented in C, and their practical running times were tested, to establish which performs better in practice.

2: Suffix array

The suffix array is a simplified data structure of the suffix tree, as we can from any suffix tree construct a corresponding suffix array [3]. A array is defined as a lexicographical ordering of all suffixes of a string. We can represent the suffix array as an array of the indices to the start of each suffix.

$$\text{SA(s)} = [\text{suffix}_{i_1}(s) \leq \dots \leq \text{suffix}_{i_n}(s)], \; 1 \leq i \leq n \tag{3}$$

For simplicity, an visual example of all of the unsorted suffixes, and the respective suffix array is shown in figure 1, with an accompanying suffix tree.

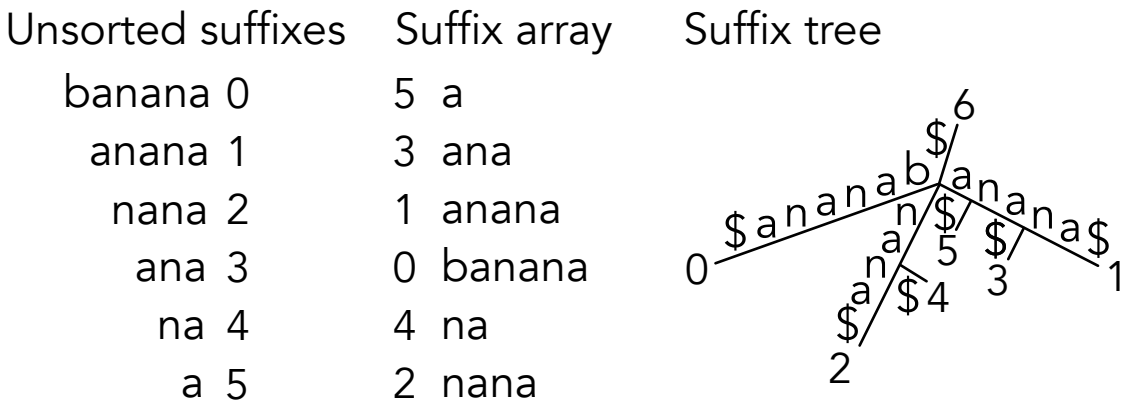

Figure 1: Visual example of a suffix array and a suffix tree of the string 'banana'.

A suffix tree is defined as a compressed trie of all the suffixes of a string. Therefore like the suffix array, it describes patterns in the string. The benefit of making a suffix array versus making a suffix tree, is the memory space requirement for both of them. Both of them use linear space, but suffix trees require 1 node per element, so it uses more bytes per element. With suffix arrays we only require an array of integers for each element. So it can be a question of using 16 to 32 bytes per character with a suffix tree, and instead only using 4 bytes for each character with a suffix array.

The naive suffix array construction algorithm is very straightforward. We simply follow the definition and sort all of the suffixes in lexicographical order. The running time of this algorithm is straight forward as well. The algorithm needs to compare two suffixes and determine which of them is more lexicographically significant. This means we can at worst case compare $O(n)$ characters for each comparison. The algorithm also depends on a sorting algorithm. The running time of the applied sorting algorithm also applies for suffix array construction. If using efficient comparison sorting algorithms, like merge sort and quick sort, we can achieve $O(n^2 \log n)$. Applying linear time sorting algorithms like bucket sort and radix sort, brings the running time to $O(n^2)$, which is the best you can achieve with this approach.

3: Sanders suffix array construction

To achieve linear time suffix array construction there are multiple approaches. One way is to construct a suffix tree in linear time, using McCreights or Ukkonen's algorithms [2, 4], and construct the suffix array in linear time using the suffix tree. This however relies on having one of these implementations, and it still requires the same amount of memory as a suffix tree during construction, which is against the main benefit of directly constructing a suffix array.

The general idea of making a linear suffix array construction algorithm, is applying a divide and conquer approach to the problem. For example we can divide the suffix array into two subproblems, each containing every other value. Sanders and Kärkkäinens algorithm uses a such an approach.

Sanders and Kärkkäinen proposed an algorithm, where they divide the problem into two one-third and two-thirds arrays, sort each of the arrays independently and finally merge the two arrays into the suffix array [1]. Their algorithm is commonly referred to as the skew algorithm, as two array sizes are skewed. The two arrays have suffixes divided among them based on the results of suffix index i mod 3. The first smaller array contains all suffixes that follow $i \bmod 3 = 0$, and the second larger array contains all suffixes that follow $i \bmod 3 \neq 0$. Therefore the arrays are respectively referred to as SA^0 and SA^{12}, based on the i mod 3 results the suffix indices give.

The algorithm, can be divided into 3 discrete steps: sorting of SA^{12}, sorting of SA^0 and merging the arrays. Figure 2, contains a visual example of the algorithm.

Step-1 SA^{12}: Sort the SA^{12} array using a stable linear sorting algorithm by the 3 first characters of the corresponding suffixes. If the suffix is at the end, and we need to sort by characters beyond the last character of the string, we assign a special character (typically denoted as \$) to these positions, which has a priority preceding all other characters. Confirming whether the sorting of SA^{12} is correct, when sorting with 3 characters, we assign lexicographical (lex) names to all of the size 3 prefixes of sorted suffixes. If these are all unique, we have confirmed that SA^{12} is sorted, and we continue to **step-2**. If they are not unique, we are uncertain if they are correctly sorted. Therefore we construct a new string, using the lex numbers as characters with the following pattern:

$$\text{New string} = \text{"suffixes } i \bmod 3 = 1\text{"} \ \# \ \text{"suffixes } i \bmod 3 = 2\text{"} \tag{4}$$

It is crucial that the suffixes indices are collected in ascending order, and we assign their respective lex numbers in this order to the new string, on either side of the special character #. The

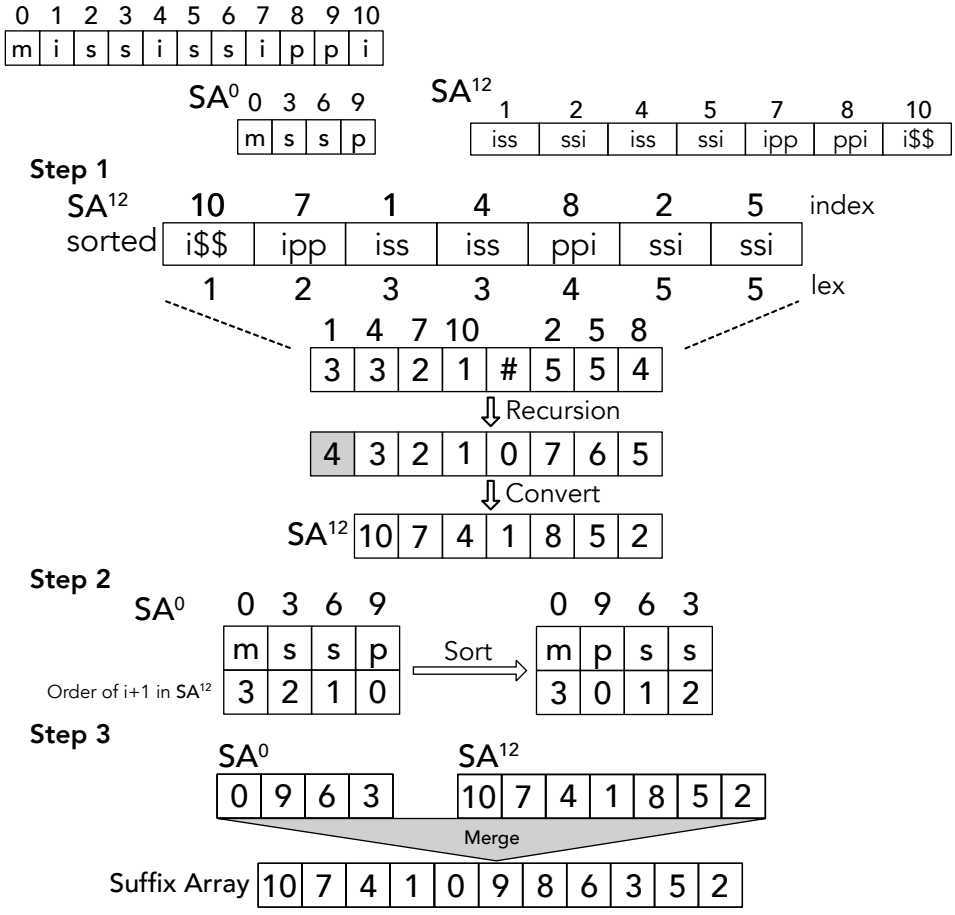

Figure 2: Visual example of the Sanders and Kärkkäinen algorithm, on the input string mississippi. Visual aid for the explanation of the algorithm.

special character is just a separator between the two sides of the string, which is ordered first lexicographically to all lex numbers. With the newly constructed string, we do a recursive call using the same construction algorithm. It goes through the same steps: we get a suffix array of the new string and construct the SA^{12} array by getting the corresponding suffix index, which corresponds to the lex number referenced in the suffix array. With the SA^{12} sorted, we proceed to **step 2**. The running time in this step linear $O(n)$, because we use a linear sorting algorithm, and we only sort with a constant number of characters (3), and assigning lex-names is also linear as we iterate through

each suffix, and check if its identical to the preceding suffix. The running time of the recursion is discussed at the end of this section.

Step-2 SA^0: Sort the SA^0 array using a stable linear sorting algorithm by the first character in the corresponding suffixes, and then by the ordering of the next suffix in SA^{12}. This can be done in one go, by sorting them as a pair $(char[i], \text{order of } i+1 \text{ in } SA^{12})$, or by first sorting them by character and then checking whether they are lexicographically unique, if not we sort those who are not unique by order of $i+1$ in SA^{12}. With SA^0 sorted, we proceed to **step 3**. The running time of this step is linear $O(n)$, as we sort with a linear sorting algorithm with a constant number of characters.

Step-3 Merge: The last step is to merge the sorted SA^0 and sorted SA^{12} into the suffix array. The merging is straightforward, as we can iteratively compare suffixes from SA^0 and SA^{12} and determine which of these two should be the next entry into the suffix array, increment forward in the array chosen from, and repeat. Due to how the arrays are separated, we can use the ordering in SA^{12} to determine cases where the first characters from SA^0 and SA^{12} match. There are two separate cases, for $i \bmod 3 = 1$ and $i \bmod 3 = 2$.

Case $i \bmod 3 = 1$ *in* SA^{12}*:* In the case of comparing a index from SA^0 to one from SA^{12} where $i \bmod 3 = 1$. We first compare the corresponding first character of the suffixes they correspond to. If they are a mismatch, we take the lexicographically smaller value and store it in the suffix array. If they match, we compare the order of the next suffix (+1) in SA^{12}, and store suffix which the smaller order in the suffix array. In short we compare $(s[i], \text{order of } i+1 \text{ in } SA^{12})$.

Case $i \bmod 3 = 2$ *in* SA^{12}*:* In the case of comparing a index from SA^0 to a index from SA^{12} where $i \bmod 3 = 2$. We first compare the first character of the corresponding suffixes, if still undecided we compare the second character of the suffixes. If decided we add the lexicographically smaller index into the suffix array, and if we are still undecided we compare the order of the next suffix (+2) in SA^{12}. In short we compare $(s[i], s[i+1], \text{order of } i+2 \text{ in } SA^{12})$.

When one of the arrays SA^0 or SA^{12} have been fully added into the suffix array, the remainder of the other array is put at the end of the suffix array. After this the arrays have been merged, we have the suffix array and the algorithm is finished.

The running time of the merging is linear, as for each suffix pair, we do a constant number of comparisons. As discussed at each step, the running time is linear $O(n)$. However the uncertain part of the algorithm is the recursion, which follows the recurrence $T(n) = O(n) + T(2/3n), T(n) =$

$O(1)$ for $n < 3$. In the master theorem, according to *Introduction to Algorithms* by Cormen, et. al [6], with these constants we find that the running time is proportional to $O(n)$, due to $f(n) = O(n^{\log_{2/3} 1}) = O(n^0)$, therefore the $T(2/3n)$ is proportional to $O(1)$, and we only have $O(n)$ remaining. Therefore the final running time is $O(n)$.

4: Implementations

Both algorithms were implemented in C, using only standard libraries, so they can be installed in any system with a standard C compiler. Both programs use the same input setup, where they read in a file and read the entire contents of the file as the input string for the algorithm, and then they print out the suffix array.

The implementation of the naive suffix array algorithm is very simple and straightforward. We get the read-in string, make an array of pointers to the start of all the suffixes. The array is fed to the built-in quick-sort (qsort) function in C, and the strcmp function from the string.h library is used as the comparison function to determine which suffix is lexicographically smaller. Finally convert the pointer addresses into the corresponding index values, and the suffix array is complete.

For the implementation of the Sanders suffix array construction algorithm, there are many particular details that need to be outlined as the algorithm is described in a general way, which leaves out specific points needed when implementing the algorithms.

Determining the order from SA^{12}, should be done with a constant look up, instead of doing a linear search $O(n)$ through the SA^{12} which would make parts of the algorithm $O(n^2)$. By making an inverse array of SA^{12}, we can look up in this array using the suffix index and get the value of the its order in SA^{12}.

Construction of the lex-number string is not specified in the algorithm, but the same kind of trick of using an inverse array can be used. The suffix index from SA^{12} corresponds to indices to the inverse array, and the contents are the corresponding lex-numbers of the suffix. By traversing this array in order and retrieving the lex-number, and storing which suffix corresponds to this lex-number, in a separate array. When the suffix array is retrieved from the recursive call, we can use the suffix indices of the new string, and look up which suffix gave this lex-number, and reconstruct SA^{12}.

To ensure the algorithm has a linear running time, it needs a linear running time sorting algo-

rithm. For this purpose a linear sorting algorithm was implemented. This process went through several iterations, due to later realizations and optimization of memory and running time. The sorting algorithm seems to be the major source of the running time and memory inefficiency.

First a simple Bucket sort was implemented, using the integer of the char as the means of sorting, and therefore it used 256 buckets. However this algorithm was not suited for the new constructed strings from the recursion, where the alphabet size can grow up size of SA^{12} so it becomes depended on n.

Therefore a Radix sort was instead implemented, by sorting the integers as sections of 4 bits using bitwise operations, and therefore divided the values into 16 buckets. This system worked, but the buckets were constructed as arrays, that got extended when full, by doubling the size. However this had a massive memory consumption for large input, and the buckets were restructured as linked lists, with a start node which pointed to end of the list and the start of the list, so new nodes could be added in constant time. However allocating a node every time they were needed made the running time much slower than before. So instead all of the nodes were allocated at the start, as we know there are as many nodes as elements we are sorting. This kept the memory consumption the same, but the running time was much faster than before.

Testing the correctness of the algorithms, was done by getting naive implementation first to work, and test whether it produced correct suffix arrays, by comparing to known suffix array results and examples derived by hand. Afterwards, to test whether the sanders implementation worked, a range of test cases of differing sizes and complexity were collected, and the outputs of both algorithms were compared, if both agreed on all cases, then they were concluded to be correct. When there was a disagreement, both outputs were compared to identify exactly what the difference was, and which output was correct. This was an important part of the debugging process.

5: Practical running times

The running time of the algorithms were tested using a variety of inputs, with increasing input sizes to see what the trends were. The different types of inputs used were worst case (all equal), random DNA sequences, simulated repetitive DNA sequences, simulated Chomsky text and randomly sampled DNA data. The running times were produced by running 10 replications for each data input, and taking the median of over the replications, to account for inherent variability that can occur in

running times.

On figure 3, we have the running times using the worst case input with both algorithms. It can clearly be seen that the Sanders algorithm performs much better on worst case data than the naive algorithm. This occurs, because the naive needs to match all of the suffixes to the end of string to determine which one has a higher priority.

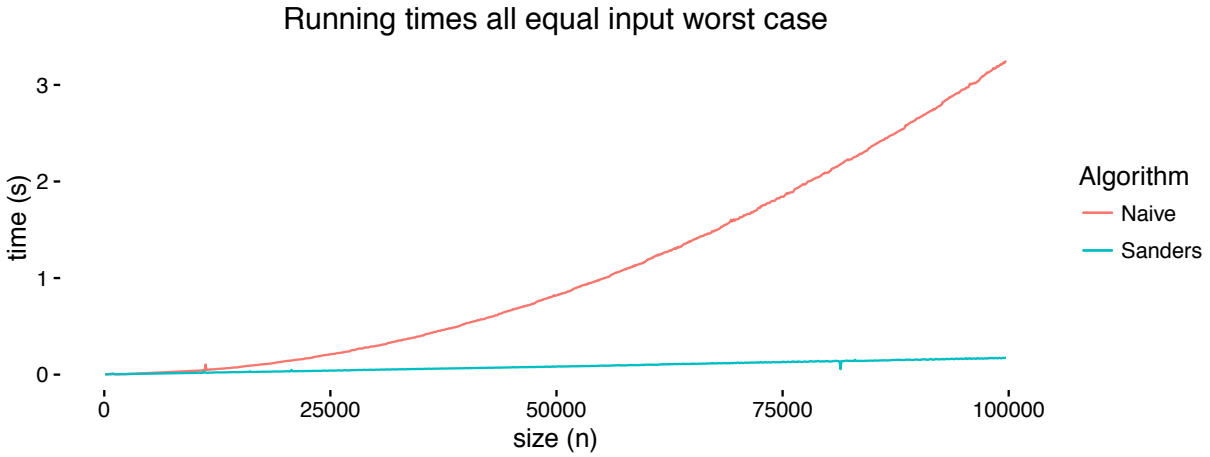

Figure 3: Worst case analysis on both inputs, using a sequence of alphabet size 1. Therefore all characters will be matched, and we get a worst case $O(n)$ comparison

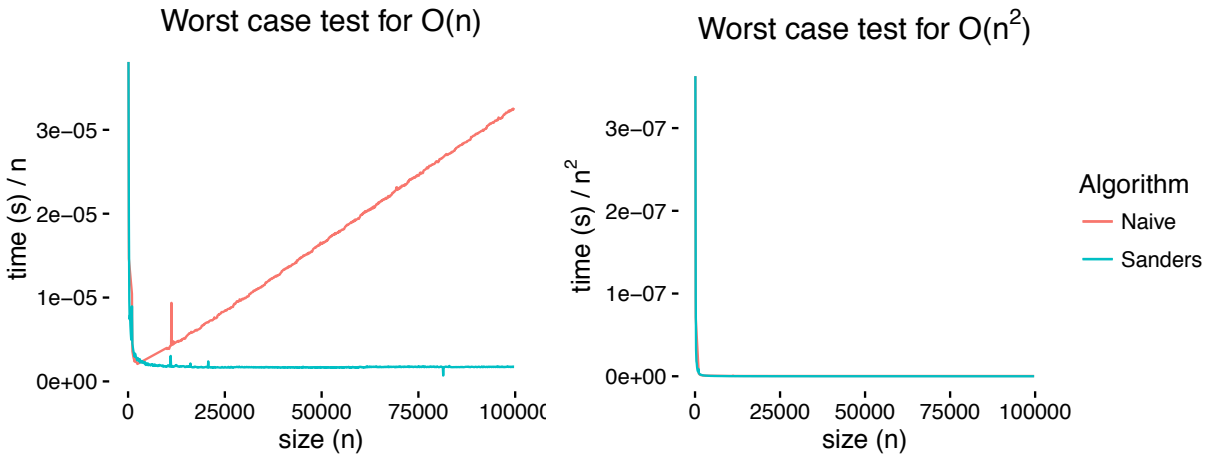

Figure 4: Test whether the running time is linear $O(n)$ on the left and squared $O(n^2)$ on the right, by dividing the time by the corresponding asymptotic factor.

To test whether the running time of the algorithms were linear, the running time is divided by the corresponding asymptotic factor. If the value is below a specific constant after a certain sample size, then we confirm its linear. This relationship can be seen in the definition of the O notation.

$$\exists\ c, N \forall\ n \geq N \cdot R(n) \leq c \cdot n \leftrightarrow \frac{R(n)}{n} \leq c \tag{5}$$

This also applies for testing $O(n^2)$. On figure 4 these two tests have been done for the worst case data, respectively. On these two figures, it can clearly be seen that Sanders is linear, and that the naive algorithm is quadratic, as naive is still linear when we test for $O(n)$ and when we test for $O(n^2)$ it stays below a constant.

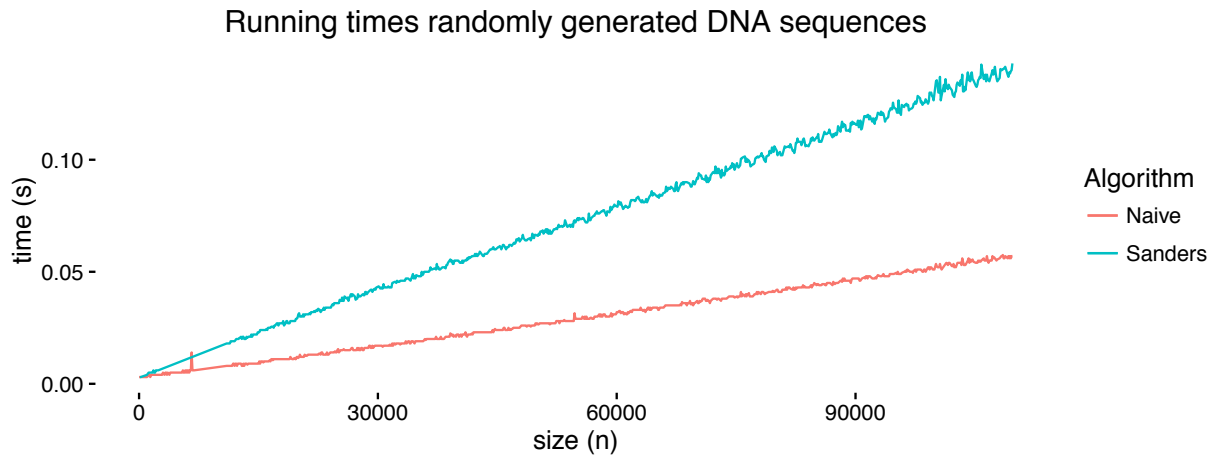

Figure 5: Running time analysis of randomly generated DNA sequences, with the two algorithms. Shows the performance of algorithms on random strings, when the alphabet size is very small.

If we look at random DNA sequences, where we have very few long repetitions in the string simply due to how unlikely it is, for example there are $4^{15} = 1.0737418 \times 10^9$ possible combinations of a DNA sequences of length 15. Therefore its very unlikely we will have many long sections repeated in the data. We can see on figure 5, that the naive performs much better than the sanders when dealing with random data like this. Due to the number of characters needed to be compared, before determining the necessary order, is very small.

If we compare a case which is not random, but there is a regular repetitive pattern in the data. For this a selection of repetitive patterns, for example like 'ATATAT' or 'GCGCGC', are randomly

generated and place together with random sizes and order. How do the algorithms react to this kind of data? Figure 6 shows these kinds of repetitive patterns affect the running time. We can see the naive having an unstable quadratic growth, but is faster that the Sanders, until a certain input size, in this case around 30000. Where as the Sanders still has a stable linear growth.

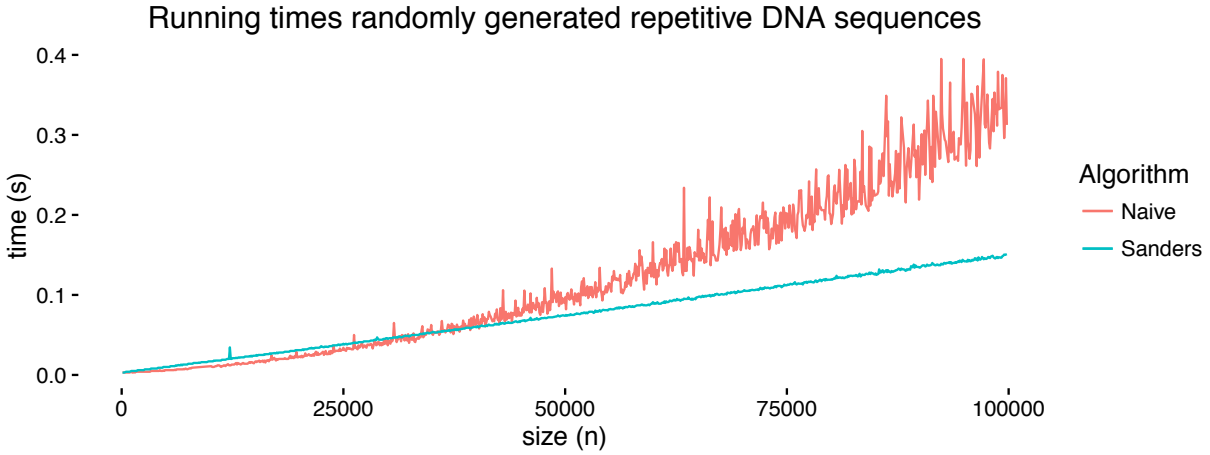

Figure 6: Running time analysis of generated repetitive DNA sequences with the two algorithms. Shows the performance, of the algorithm, when there are a lot of matching and recurring patterns in the suffix.

To test the running time more normal grammatically correct text and and have a increasing tendency for the runtime analysis, a chomsky text generator was used [7]. An example of a random sentence it can produce:

> *Comparing these examples with their parasitic gap counterparts in (96) and (97), we see that the systematic use of complex symbols does not readily tolerate problems of phonemic and morphological analysis.*

It produces text, with normal structure you would find in general English literature. Figure 7 shows the running time difference, using such generated text. We clearly see that the naive implementation is faster, than the sanders, likely due to the unlikelihood of the same pattern of words occurring often, so we do not match far into the suffixes.

All of the previous analysis, have been different simulation tests, however how do the algorithms behave on actual DNA data? Using the chromosome 1 sequence, the first 10 million base pairs, the

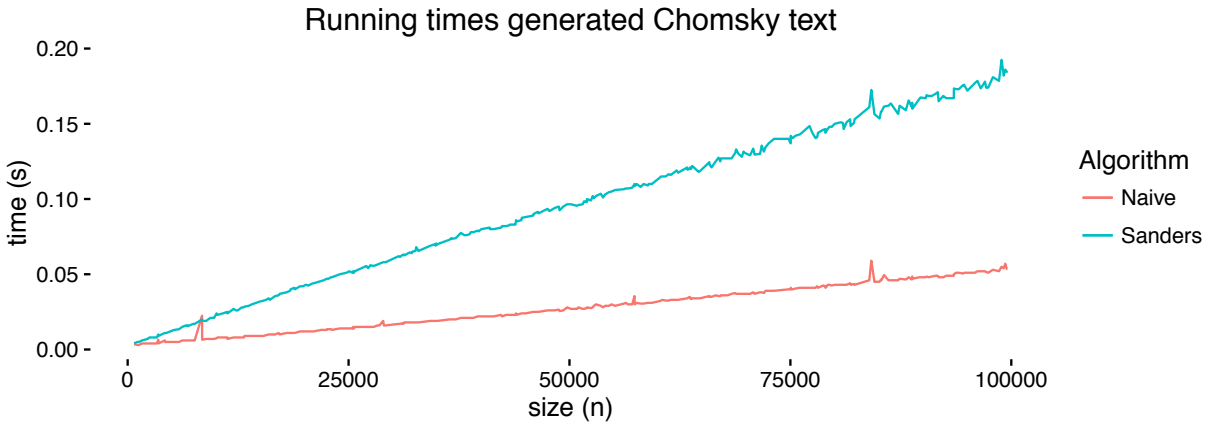

Figure 7: Running time analysis of generated Chomsky text. Shows how the algorithms perform ordinary text data, versus how they perform on sequence data.

data was randomly sampled with the increasing lengths, to see the effect on the runtime. Figure 8 shows the running time of this data. In general the naive is faster than the Sanders algorithm, however there are some spikes, and the majority are not due to random fluctuations in the CPU. However in the data, there are cases long stretches of completely N, or missing nucleotide regions. In the cases these areas were sampled, there is a worst case running time, due to them being all equal.

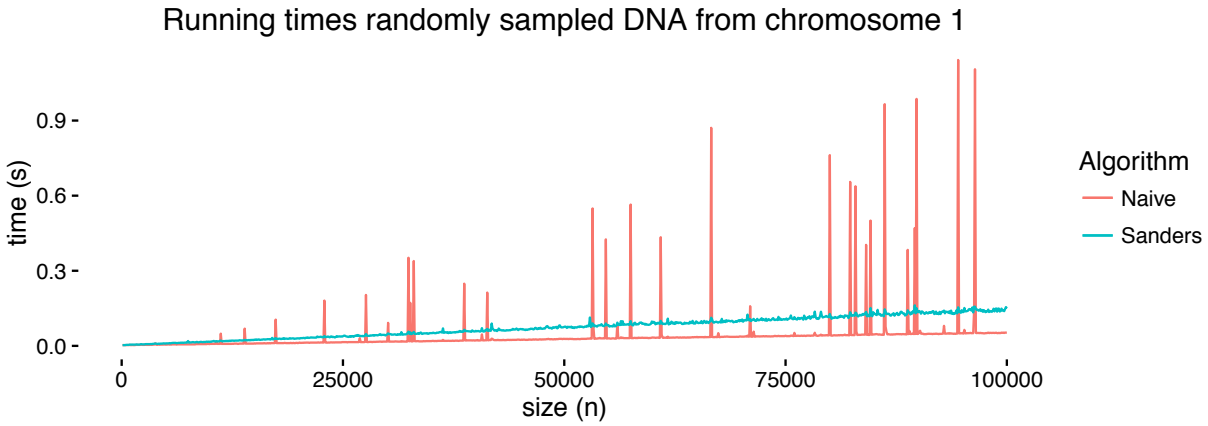

Figure 8: Running time analysis of randomly sampled DNA data sampled from chromosome 1 in the human genome.

The plan was to include analysis of the runtime of the algorithms, constructing on a whole chromosome. However there were problems in getting the algorithms to run, as there seemed to be some kind memory leak in regards to file reading very large files. Therefore the algorithms just produced segmentation faults. Due to lack of time, this problem could not be fixed.

The sanders uses the recursion step to resolve the sorting, when there is uncertainty whether it is correctly sorted with only 3 characters. An interesting question is how many steps of recursion does it need to resolve the problem? Figure 9 shows a recursion depth analysis, with 4 different types of inputs of increasing size, using the same means of generation as described for runtime experiments. We can see that even in the worst case, with all equal input, the recursion depth is very small, even with input sizes greater than 60000, we only have a recursion depth of 10. Random inputs scale very slowly, reaching down to 2, and similarly with text, which is more repetitive than random, reaching down to 4. The repetitive data, is more randomly generated and therefore unstable, so we see fluctuations in the recursion depth, but it has a similar pattern to the all equal input curve. This analysis shows, that very few recursions are needed to resolve the sorting, and the worst case input shows what upper-bound on the recursion depth is.

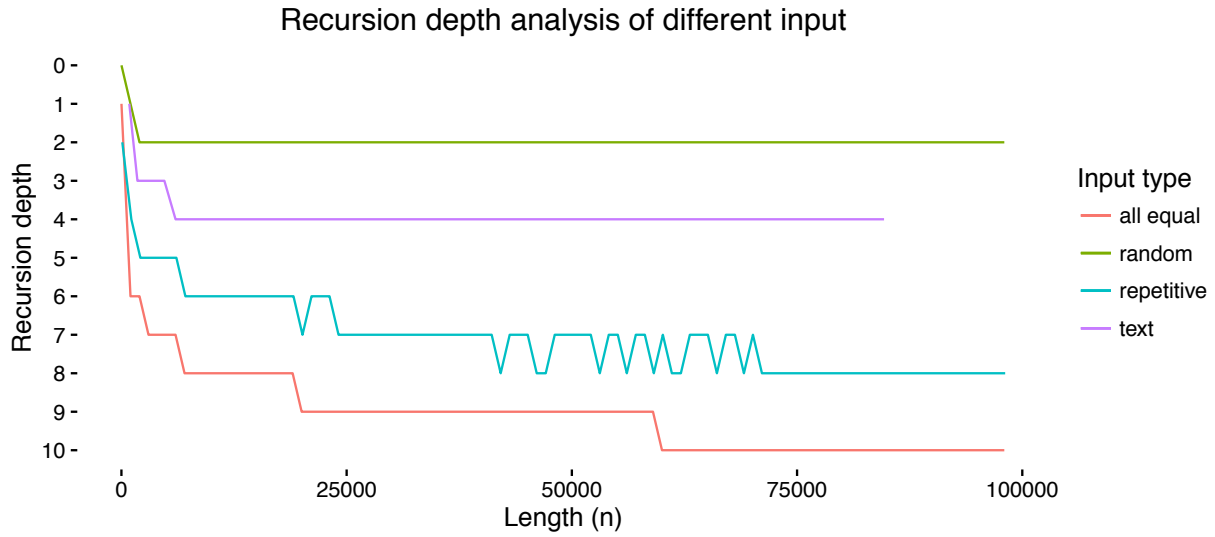

Figure 9: Recursion depth analysis, of different inputs and varying input sizes. Four different types of inputs, using the same means of generation as indicated for the running time analyses.

6: Conclusion

The running times of naive implementation heavily depends on what kind of input is given to it, where as the Sanders implementation is very stable for any given input. However the naive implementation performs better on data which these algorithms are expected to be used on, for example real DNA, and regular text. However the more repetitive the data is, the less efficient the naive implementation will be. Knowing more about specific properties of the data, such as the repetitiveness in the data, can make the choice of algorithm more clear.

References

[1] Kärkkäinen, J., & Sanders, P. Simple linear work suffix array construction. Colloq. Autom. Lang. Program. 14186, 943–955 (2003).

[2] McCreight, E. M. A Space-Economical Suffix Tree Construction Algorithm. J. Acm 23, 262–272 (1976).

[3] Manber, U. & Myers, G. Suffix arrays: A new method for on-line string searches. SIAM J. Comput., 22(5):935–948, Oct. 1993.

[4] Ukkonen, E. On-line construction of suffix trees. Algorithmica 14, 249–260 (1995).

[5] Weiner, P. Linear pattern matching algorithms. Switch. Autom. Theory, 1973. SWAT '08. IEEE Conf. Rec. 14th Annu. Symp. 1–11 (1973). doi:10.1109/SWAT.1973.13

[6] Cormen T. H. & Leiserson, C. E. & Rivest, R. L. & Stein, C. Introduction to Algorithms (Third edition), 2009, Massachusetts Institute of Technology.

[7] Uploaded by user Raymond Hettinger, Chomsky random text generator (Python recipe), http://code.activestate.com/recipes/440546-chomsky-random-text-generator/