Linear-Time Suffix Array Implementation in Haskell

Anna Geiduschek and Ben Isaacs
CS240H Final Project
GitHub: https://github.com/ageiduschek/DC3-Suffix-Array
June 5th, 2014

Abstract

The purpose of our project was to create a linear-time suffix array implementation in Haskell, with the ultimate goal of implementing the Burrows-Wheeler Transform on a given string. A suffix array for a given string *s* is in essence an array that stores all possible suffixes of *s*, lexicographically sorted. The Burrows-Wheeler Transform is a way of rearranging a string into another string in a way that is reversible. The transformed string tends to be, in many cases, more compressible than the original string, and so the Burrows-Wheeler Transform has useful applications in compression. Searching the internet, including both Hackage and other sources, we could not find any linear-time suffix array libraries. We did find a couple libraries that used naive algorithms to calculate a suffix array in $O(n^2 logn)$ time, but when using a suffix array for tasks such as compressing a large string, then this $O(n)$ implementation is asymptotically superior.

Motivation

We are both interested in data structures, and wondering how they are implemented in Haskell. Data structures are not something expounded upon in detail in CS240H, so we decided to search outside the class to find something that we could use our newly minted Haskell skills to implement a data structure. As will be discussed in later sections of this paper, the algorithms surrounding suffix arrays are often recursive, so we thought that Haskell's recursive nature would lend itself well to these algorithms.

As mentioned in the above abstract, there were no accessible linear-time suffix array implementations in Haskell available at the time of the writing of this paper. Considering the range of uses for efficient suffix array creation, we figured that this library would be useful to the greater Haskell community.

The Suffix Array

A **suffix array** for a given string *s* is an array which contains every possible suffix of *s* sorted in lexicographical order. For instance, the suffix array for the string "banana" (which is a widely used example for some reason) is **["", "a", "ana", "anana", "banana", "na", "nana"]**. Another example for the string "nonsense" would be **["", "e", "ense", "nonsense", "nse", "nsense", "onsense", "se", "sense"].** Note that the empty string, which is a suffix for any string starting immediately after the final character, is always the first element in a suffix array, but the ordering of the rest of the suffix array depends on the string. A **generalized suffix array** for a set of given strings *S* is an array which contains every suffix from every string in *S*, and also has per-element information denoting which string that suffix

came from. In the above example, we could tag every element in the suffix array for “banana” and use a different tag for every element in the suffix array for “nonsense,” and then merge them in linear time.

It should also be noted that the way that we store our suffix array internally is not with an actual array of strings but rather with a structure an array of indices where each suffix starts and the string itself. For instance, for “banana”, the suffix array could be **[6, 5, 3, 1, 0, 4, 2]**. Note that we could easily reconstruct an ‘actual’ suffix array as above by running through our index-based suffix array and retrieving the suffix that belongs at that index in the suffix array by getting the substring of the string that begins at the index given in the array. For instance, the substring starting at index 6 in “banana” is “”, the substring starting at index 5 is “a”, the substring starting at index 3 is “ana”, etc. Therefore storing an array of indices alongside the string and accessing substrings as necessary is both valid and considerably more space-efficient than storing a true array of strings.

The DC3 Algorithm

The DC3 Algorithm is a linear-time algorithm to construct a suffix array for a string, which we implemented in `SuffixArray.hs`. The algorithm uses a recursive divide and conquer algorithm to find the relative ordering of suffixes at indices, i, in our input string where i mod 3 != 0 in linear time. It then uses this information to find the relative ordering of the suffixes at indices, i, in our input string where i mod 3 == 0. Finally, it uses a simple merge algorithm on these two orderings to recover the entire suffix array.

Note: While the DC3 suffix array algorithm does run in *O(n)* time, the involved constant factor is quite high. There do exist significantly simpler *O(nlog(n))* algorithms that tend to outperform DC3 on reasonably sized inputs.

Range Minimum Queries

The Range Minimum Query problem can be defined as follows: given an array A, $RMQ_A(i, j) = k$ where $i \leq k \leq j$ and A[k] is the minimum value in the range between A[i] and A[j]. Any query $RMQ_A(i, j)$ can be solved in O(1) time with O(n) preprocessing time using a structure developed by Fisher and Heun in 2006. This data structure is layered on top of a collection of RMQ structures called sparse tables which precomputes all of the minimum values in ranges of lengths 2^k in O(nlog(n)) time. We implemented the Fischer-Heun structure in `FischerHeun.hs` and sparse tables in `SparseTable.hs`. Being able to compute RMQ in constant time with linear preprocessing time is critical for being able to solve the Longest Common Extension problem (see below) with O(n) preprocessing and O(1) query time.

Applications: Longest Common Extension

The Longest Common Extension (LCE) problem on two strings, p and q, can be defined as follows: $LCE_{p,q}(i, j)$ equals the length of the prefix that strings starting at p[i] and q[i] have in common. More generally, for *k* strings $p_1, \ldots, p_k$, we have $LCP_{p1 \ldots pk}(i_1, \ldots, i_k)$ is the length of the prefix that all substrings $p_x[i_x]$ have in common. For example, for strings p = "nonsense" and q = "tense", we have that $LCE_{p,q}(2, 2) = 3$ ("**nse**nse" and "**nse**" have a prefix of length three in common) and $LCE_{p,q}(0, 2) = 1$ ("**n**onnsense" and "**n**se" have a prefix of length one in common).

The LCE problem has a variety of applications in string processing algorithms (as discussed below in the Applications section), thus it is important to be able to compute LCE information quickly. LCE queries are used to find the longest palindromic substring in a string (this has applications in computational genomics), to perform *k*-approximate matcing (given a string S, is there a substring of S that matches a pattern P in all but at most *k* places?), and to perform wildcard matching (finding matches of pattern P in input S where P can have wildcard values).

Generalized suffix arrays are often used to solve LCE queries two strings O(1) time (more generally, for *k* strings, we can solve LCE in O(k) time) with O(n) preprocessing time. We implement the algorithm is as follows. First, create a suffix array SA in O(n) time using the DC3 algorithm. Next, generate an array LCP where LCP[i] equals the length of the longest common prefix of the suffixes SA[i] and SA[i + 1]. This step takes O(n) time using an algorithm discussed in *Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications* by Kasai, et al. in 2001. Let pos_i = position in the suffix array of the suffix p[i] and pos_j = position in the suffix array of the suffix q[j]. To solve $LCE_{p,q}(i, j)$ we merely need to know the minimum value in our LCP array between indices pos_i and pos_j. To solve this in O(1) time, we generate a Fischer-Heun RMQ structure as discussed above in O(n) time. The `GeneralizedSuffixArray` data type stores the LCP array and Fischer-Heun structure, so our function

```
lce :: GeneralizedSuffixArray -> [Index] -> IO Int
```

performs LCE queries on the input strings in constant time.

Application: Burrows-Wheeler Transforms

A Burrows-Wheeler Transform (BWT) is a reversible permutation on a string that tends to group identical characters next to each other, allowing the string to be more easily compressed. To construct the BWT on a string S, we create a lexicographically sorted matrix of all the rotations of S with an end of string character appended (often '$'), and return the last column of that matrix (note, the EOF character is ranked less than all other characters). For example, the input "banana" has the following matrix:

$ b a n a n **a**
a $ b a n a **n**

a n a $ b a **n**
a n a n a $ **b**
b a n a n a **$**
n a $ b a n **a**
n a n a $ b **a**

Thus, the BWT of “banana$” is “annb$aa”. Notice that if we add one to the index of each character in our BWT, we get the indices of the characters in the first column of this matrix, which also happens to be the column represented by a suffix array. Thus, we can actually create a BWT in O(n) time by creating a suffix array, subtracting one from each index in our suffix array, then return those characters at the subtracted indices.

The BWT might seem like magic, but it works due to the predictable nature of text, which has repetitive patterns of characters, often repeated words, etc. The results become even more dramatic as the piece of text grows. For example, look at the output from running our `toBurrowsWheeler` function on the following excerpt from Jane Austen’s *Pride and Prejudice*:

```
ghci> toBurrowsWheeler "It is a truth universally acknowledged, that a single man in
possession of a good fortune, must be in want of a wife.However little known the feelings
or views of such a man may be on his first entering a neighbourhood, this truth is so
well fixed in the minds of the surrounding families, that he is considered as the
rightful property of some one or other of their daughters." '$'

".fhgtsfydtysrtgesldatnndetehereane,antyssreeesrnleasffe,,nnsff,sahrnoaedsdse.$
sfmmw hhdm  hua eoeeo iennhbhllhbmnhhnfrxglfnhew vhdttvpiswiooooooo i    tnndiuin nctttttgt
ttttt trggslvwerm   mrdslsef  h hlnfc lutgwie eal   oa  oaaoiwiiuiou iiiiukkoaesoo      s
i cghr  fp rbHnno  ieeooeu eprueeioettiiiwdgiaierrsn s  oeur  assaInnhhuu       o t
irrsafot osmrrei   o ooeilat"
```

The BWT can be compressed with a run-length encoding (which replaces each run of identical characters with a number and that character), or paired with move-to-front encoding and huffman coding (this is how the common bzip2 compression algorithm works).

Challenges

The biggest challenge we faced was adhering to the (self-imposed) linear time bound. When we started the project, we didn’t consider the the fact that you can’t do constant time random access in normal Haskell lists. The ability to do this is crucial to making DC3 run in O(n) time, because it relies on a variety of operations that involve using the value of one array to index into another array. To solve this problem, we ended up using mutable arrays (Data.Array.IO), which forced us to work in the IO monad for much of the program. Ultimately, this was a learning experience for the both of us, because we realized that many data structures algorithms are developed based on imperative programming paradigms.

It would be very interesting to see if anyone can develop an algorithm for building suffix arrays in O(n) time bounds based on purely functional code.

Next Steps

If we have a suffix array augmented with LCP information and an RMQ structure built on top of the LCP array (as we created in our project), then we can actually also construct a suffix tree on the input in linear time. A suffix tree is a tree that encodes all of the suffixes in an input string. A suffix tree on a single string can be used to efficiently find if (and where) the input string contains a given pattern or to find the longest repeated subsequence in a string. A generalized suffix tree (made from a generalized suffix array) can be used to efficiently solve the longest common subsequence problem on two or more strings. The algorithm for creating a suffix tree out of an augmented suffix array is non-trivial and we didn't have time to get to it, but it would be have been an interesting extension to this project.

API

GitHub: https://github.com/ageiduschek/DC3-Suffix-Array

```
SparseTable.hs:
    -- Returns function that can be used to find RMQ over two indices on the array.
    --         Calls to sparseTableRMQ run in O(nlog(n)) time with O(1) time queries
    --         to the ouput function.
    type SparseTable = Index -> Index -> IO Index
    sparseTableRMQ :: [Int] -> IO SparseTable

FisherHeun.hs:
    -- Returns function that can be used to find RMQ over two indices on the array.
    --         Calls to fischerHeunRMQ run in O(n) time with O(1) time queries
    --         to the ouput function.
    type FischerHeun = Index -> Index -> IO Index
    fischerHeunRMQ :: [Int] -> IO FischerHeun

SuffixArray.hs:
    -- Functions to create SuffixArray types which can then be passed around:
    createSuffixArray :: String -> IO SuffixArray
    createGeneralizedSuffixArray :: [String] -> IO GeneralizedSuffixArray

    -- Returns a lexicographically ordered list of indices which each
    --        represent a suffix in the original input string.
    getSuffixRankings :: SuffixArray -> [Int]

    -- Returns a lexicographically ordered list of (string number, index)
    --        pairs which each represent a suffix in one of the original
    --       input strings.
    getGeneralizedSuffixRankings :: GeneralizedSuffixArray -> [(Int, Int)]

    -- Retrieve the original input string(s) from a suffix array.
```

```
getInputStr :: SuffixArray -> String
getInputStrs :: GeneralizedSuffixArray -> [String]

-- For demonstrative purposes.
printSuffixes :: SuffixArray -> IO ()
printGeneralizedSuffixes :: GeneralizedSuffixArray -> IO ()

-- For Generalized suffix array with input strings S_1 to S_n, accepts
--        an array of indices [i_1, …, i_n] and returns the length of
--        the shared prefix between S_1[i_1], ... , S_n[i_n] . The array
--        of indices must be the same length as the array of input
--        strings. Example usage:
--            ghci> gsa <- createGeneralizedSuffixArray [“anna”, “banana”]
--            ghci> lce gsa [0, 1]
--            2
--            ghci> lce gsa [2, 3]
--            0

lce :: GeneralizedSuffixArray -> [Index] -> IO Int

-- Takes a string and an EOF character that doesn’t appear anywhere in the
--       and returns the Burrows Wheeler transform on that string. To recover
--       the original string, pass the result of this function to
--       fromBurrowsWheeler with the same EOF character. Example usage:
--            ghci> bwt <- toBurrowsWheeler "Today you are you! That is truer than
--            true!" '$'
--            ghci> bwt
--            "!!utrnsyeeu $h hdoruutT aTyyeattia   oorra  "

toBurrowsWheeler :: String -> Char -> IO String


-- Takes a string that has been transformed by toBurrowsWheeler and recovers the
--       original string. Must use the same EOF character that was given to
--       toBurrowsWheeler. Example usage:
--            ghci> str <- toBurrowsWheeler "!!utrnsyeeu $h hdoruutT aTyyeattia
--            oorra  " '$'
--            ghci> str
--            "Today you are you! That is truer than true!"
fromBurrowsWheeler :: String -> Char -> IO String
```