2009 Haskell January Test
String Matching and Suffix Trees

This test comprises four parts and the maximum mark is 25. Parts I, II and III are worth 24 of the 25 marks available. One bonus mark is available for an optional question in Part II. This will be added to your total, although your final mark will be capped at 25. The **2010 Haskell Programming Prize** will be awarded for the best attempt(s) at Part IV.

Credit will be awarded throughout for clarity, conciseness, *useful* commenting and the appropriate use of Haskell's various language features and predefined functions.

WARNING: The examples and test cases here are not guaranteed to exercise all aspects of your code. You may therefore wish to define your own tests to complement the ones provided.

1 Introduction

String matching is a fundamental problem in computer science with many wide ranging applications including natural language processing, data compression, and biological sequence analysis. The *exact* string matching problem asks whether a given search string appears in a (usually much larger) text string and, if so, where. As an example, the string "i miss mississippi" contains six occurrences of the (sub)string "s" (at indices 4, 5, 9, 10, 12, 13), three of "is" (indices 3, 8, 11), two of "issi" (indices 8, 11), and so on. Note that indices are numbered from 0.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
text	i		m	i	s	s		m	i	s	s	i	s	s	i	p	p	i
s					↑	↑				↑	↑		↑	↑				
is				↑					↑			↑						
issi									↑			↑						

The simple way to search a text string t for a search string s is to work from left to right along t, analysing all its *suffixes*. If s is a *prefix* of any of these suffixes, then s is contained within t. For example, the suffixes of "banana" are:

0	"banana"
1	"anana"
2	"nana"
3	"ana"
4	"na"
5	"a"

As an example, note that "an" appears twice in "banana" as "an" is a prefix of both "anana" and "ana"; the associated indices are 1 and 3.

The problem with this "naive" approach is that the worst-case search time is proportional to the product of the lengths of s and t (if you can't see why don't worry!). In order to improve matters a number of very ingenious data structures have been invented to speed up the search. One of these is called a *suffix tree* and is the subject of this exercise. Once a suffix tree has been built, it is possible to determine whether s is a substring of t in a time that is proportional to the length of s alone. Many other search problems are also much more efficient when performed over suffix trees.

2 Suffix Trees

A suffix tree represents a text string. The component characters of a string, s say, are indexed $0...n-1$ where n is the length of s. Suffix trees are built from internal nodes and leaf nodes. Each leaf of the suffix tree for s contains an integer index into s. Each subtree of an internal node is labelled with a string. The tree has the property that the concatenation of the subtree labels on a path from the root of the tree to a leaf represents exactly one suffix of s; the index stored at the leaf is the index within s of that suffix. A path that visits the subtree labels $s_1, s_2, ..., s_k$ will be denoted $s_1 \to s_2 \to ... \to s_k$. Note that all suffixes are unique as they have different lengths. As an example, Figures 1 and 2 show the suffix trees for the strings "banana" and "mississippi".

Notice that each internal node represents a *repeated* substring. For example the node marked in black in Figure 2 corresponds to the path "s"→"si", i.e. the substring "ssi". This substring appears twice in "mississippi", specifically in the suffixes "ssissippi" (index 2) and "ssippi" (index 5). Hence the two leaf nodes, and associated labels, that emanate from the black node. Notice that the same path tells us that the substring "ss" also appears at indices 2 and 5, by virtue of

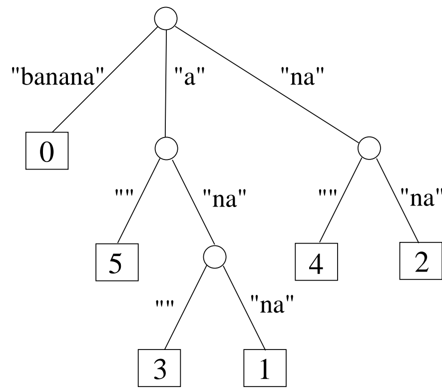

Figure 1: Suffix Tree for “banana”

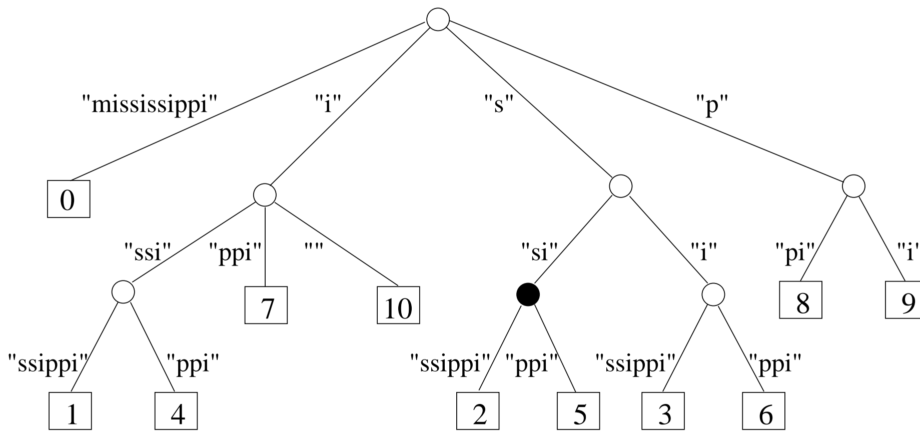

Figure 2: Suffix Tree for “mississippi”

the fact that “ss” is a prefix of the concatenated path labels “ssi”; this implies that “ss” is a prefix of the same suffixes “ssissippi” and “ssippi”.

As in the naive scheme, string matching on a suffix tree works by searching for prefixes of suffixes, but now the problem is reduced to a form of tree search.

Consider the suffix tree for some text string t and suppose that we are searching for any occurrences of the substring s. We begin by comparing s with the strings that label each subtree referenced from the root node of the tree. Consider one such subtree with label a, say. There are three cases, which should be tested in order:

1. If s is a prefix of a then it is a prefix of *every* suffix indexed in the leaves of that subtree. For example, in Figure 2, if s=“p” and we are inspecting the subtree labelled “p”, we see that “p” is a substring of “p” – a string s is trivially a substring of itself. Looking down, this tells us that “p” is a prefix of both “ppi” and “pi” and therefore a substring of “mississippi”. The required indices are thus located in the leaf nodes reachable from the “p” subtree, i.e. 8 and 9.

2. If a is a prefix of s then the search must continue into the subtree, after removing the prefix a from s. For example if s=“iss” and we are inspecting the subtree labelled “i” we see that

the "i" is a prefix of "iss". We thus continue the search recursively in that subtree, using the *residual* string "ss", i.e. "iss" with the prefix "i" removed. The same rules now apply. Continuing the example, "ss" is now found to be a prefix of the subtree label "ssi" (case 1) which tells us that the original search string "iss" is a prefix of every suffix defined by the "ssi" subtree, i.e. "ississippi" (index 1) and "issippi" (index 4).

3. If neither s nor a is a prefix of the other, then the match fails at that subtree and we proceed to the next subtree, if there is one.

If none of the subtrees of a node satisfy conditions 1 or 2 then s does not occur in t. For example, with the search string "ssify" we follow the path "s"→"si" and end up comparing the residual string "fy" with the subtree labels "ssippi" and "ppi" where both matches fail.

2.1 Suffix Trees in Haskell

A suffix tree can be represented in Haskell by the following data type:

```
data SuffixTree = Leaf Int | Node [(String, SuffixTree)]
                  deriving (Eq,Ord,Show)
```

Note that each internal `Node` may have an arbitrary number of subtrees, so they are collectively stored as a list; each subtree is labelled with its associated `String`. The index stored in each leaf is an `Int`. As an example, here are the representations of the above suffix trees for "banana" and "mississippi":

```
t1 :: SuffixTree
t1
  = Node [("banana", Leaf 0),
          ("a", Node [("na", Node [("na", Leaf 1),
                                   ("", Leaf 3)]),
                      ("", Leaf 5)]),
          ("na", Node [("na", Leaf 2),
                       ("", Leaf 4)])]

t2 :: SuffixTree
t2
  = Node [("mississippi", Leaf 0),
          ("i", Node [("ssi", Node [("ssippi", Leaf 1),
                                    ("ppi", Leaf 4)]),
                      ("ppi", Leaf 7),
                      ("", Leaf 10)]),
          ("s", Node [("si", Node [("ssippi", Leaf 2),
                                   ("ppi", Leaf 5)]),
                      ("i", Node [("ssippi", Leaf 3),
                                  ("ppi", Leaf 6)])]),
          ("p", Node [("pi", Leaf 8),
                      ("i", Leaf 9)])]
```

3 What to do

The exercise is broken down into four parts. Most of the marks are assigned to the first three parts, so you are advised to complete these before moving on to Part IV.

The constants `s1` and `t1` defined in the template denote the string "banana" and its suffix tree respectively. Similarly, `s2` and `t2` for "mississippi". These can be used for testing.

PART I – Basic String Processing

The first part of the exercise requires you to write some simple string processing functions that will be useful later on.

1. Define a function `isPrefix :: String -> String -> Bool` that returns `True` iff the first given string is a prefix of the second. For example, `isPrefix "has" "haskell"` and `isPrefix "" "haskell"` should both return `True`, whilst `isPrefix "haskell" "has"` and `isPrefix "ask" "haskell"` should return `False`.

   [1 mark]

2. Define a function `removePrefix :: String -> String -> String` that returns the result of removing a prefix (first argument) from a given string (second argument). A precondition is that the first string is a prefix of the second. For example, `removePrefix "ja" "java"` should return `"va"` and `removePrefix "" "java"` should return `"java"`.

   [1 mark]

3. Define a function `suffixes :: [a] -> [[a]]` that returns the lists of all suffixes of a given string in descending order of length. For example, `suffixes "perl"` should return `["perl","erl","rl","l"]` in that order. Hint: one way to do this is by repeated application of the `tail` function.

   [4 marks]

4. Use the functions `isPrefix` and `suffixes` to define a function `isSubstring :: String -> String -> Bool` that returns `True` iff the first string is a substring of the second. For example, `isSubstring "ho" "python"` should return `True` and `isSubstring "thong" "python"` should return `False`.

   [3 marks]

5. Use the functions `isPrefix` and `suffixes` above to define a function `findSubstrings :: String -> String -> [Int]` that returns the indices of all occurrences of a given substring (first argument) in a given text string (second argument) using the 'naive' approach (Section 1). If there are no occurrences the function should return `[]`. For example, `findSubstrings "an" "banana"` should return `[1,3]` in some order and `findSubstrings "s" "mississippi"` should return `[2,3,5,6]` in some order.

   [3 marks]

PART II – String Matching on Suffix Trees

You are now going to re-implement the above substring search function, but this time with the text string represented as a suffix tree.

1. Define a function `getIndices :: SuffixTree -> [Int]` that returns the values (indices) stored in the leaves of the given suffix tree. The order in which those indices are returned is unimportant. For example, `getIndices (Node [("",Leaf 4),("",Leaf 1)])` should

return `[1,4]` in some order and `getIndices t1`, where `t1` is the suffix tree for “banana”, should return `[0,1,2,3,4,5]`, in some order.

[3 marks]

2. Using the search method described in Section 2, use `getIndices` to define a function `findSubstrings' :: String -> SuffixTree -> [Int]` that returns the indices of all occurrences of a given substring in a given suffix tree. If the search reaches a leaf node, `Leaf i`, say when the residual search string is empty (`""`) (e.g. when searching for “pi” in “mississippi”) then the match succeeds and the required (single) index is the integer stored at the leaf (result `[i]`). If the residual string is non-empty (e.g. when searching for “ippity” in ‘mississippi”) then the match fails and the result is the empty list (`[]`). For example, `findSubstrings' "an" t1` and `findSubstrings' "s" t2` should return `[1,3]` and `[2,3,5,6]` respectively, as per `findSubstrings` in Part I. Again, the order in which the elements appear is not important.

   Hints: A simple way to do this is to use the functions `isPrefix` and `removePrefix` defined earlier. This is not the most efficient solution as it requires you to scan the same string twice to check and remove a common prefix, but is sufficient to get the available marks. Note also that at each internal node you could check each subtree in turn looking for labels that match either Case 1 or Case 2 above, or you could check them *all* (`concat(Map)`?), with each returning a list of matching indices. If you do it this way then at most one such list can be non-null.

   ### Optional Simplification (Bonus mark)

   You can avoid the need to use `isPrefix` and `removePrefix` in combination by defining an additional function `partition :: Eq a => [a] -> [a] -> ([a], [a], [a])` that will extract a common prefix from two given strings, returning the prefix and the result of removing the common prefix from the two strings. For example, `partition "happy" "haskell"` would return `("ha","ppy","skell")`. It turns out that this function can also be used to simplify the building of a suffix tree (Part III). A bonus mark will be awarded for solutions that successfully make use of this function.

   [5 marks + optional bonus]

PART III – Building a Suffix Tree

The final part requires you to build a suffix tree for a given text string. The straightforward way to do this is to add each suffix of the text string in turn, starting with the longest (i.e. the text string itself). This is not necessarily the most efficient method, but is sufficient for this exercise. The function to build the tree this way is given in the template file, viz.:

```
buildTree :: String -> SuffixTree
buildTree s
  = foldl (flip insert) (Node []) (zip (suffixes s) [0..])
```

Your job is to define the function `insert :: (String, Int) -> SuffixTree -> SuffixTree` which inserts a given suffix (`String`), with its associated index (`Int`), into a given suffix tree. You may find the function easier to write in terms of the optional `partition` function outlined above.

Hints: To insert the pair (s, n) into a node, you need to inspect each labelled subtree in turn. Consider the subtree t with label a. If s and a have no prefix in common, then a and t are unchanged and the insertion progresses to the next subtree. If s and a do have a common prefix, p say, then there are two cases. If $p = a$ (i.e. a is a prefix of s) then the insertion is pushed into t using the residual substring $s - p$, i.e. s with the common prefix p removed. If $p \neq a$ then a is replaced with p and t is replaced with a new node with exactly two subtrees: one labelled $s - p$, leading to a leaf with index n, and the other being t relabelled with $a - p$. If no subtree label has a common prefix with s then the list of subtrees is extended by adding a new leaf node labelled s with index n. Note that the order of subtrees in a node is not significant.

If `insert` is correctly defined, then `findSubstrings' s (buildTree s1)` should yield the same set of indices (although possibly in a different order) as `findSubstrings' s t1` for all strings `s`. Similarly for `s2` and `t2`.

[4 marks]

Part IV – Longest Repeated Substring (Prize Question)

The longest *repeated* substring of a given text string is the longest substring that occurs two or more times (possibly overlapping) in the given text. For example, the longest repeated substring of "mississippi" is "issi". Note that the substring "is" also appears twice, for example, but "issi" is the longer of the two. Thus, define a function: `longestRepeatedSubstring :: SuffixTree -> String`. If there is more than one longest repeated substring, e.g. "miss" and "issi" in "miss mississippi", then any of them is acceptable.

[1 mark]