15-451 Algorithms
Fall 2012
D. Sleator

Suffix Trees and Suffix Arrays December 4, 2012

References:
Algorithms on Strings Trees and Sequences by Dan Gusfield
http://courses.csail.mit.edu/6.897/spring03/scribe_notes/L10/lecture10.pdf
http://courses.csail.mit.edu/6.897/spring03/scribe_notes/L11/lecture11.pdf

Outline:

- Suffix Trees
  - definition
  - properties (i.e. O(n) space)
  - applications
- Suffix Arrays
  - definition
  - how to compute a suffix array (and prefix length array) in O(n log^2 n) time
  - how to convert this into a suffix tree in O(n) time

---

## Suffix Trees

Consider a string s of length n (long). Our goal is to preprocess s to allow various kinds of queries on the string to be done efficiently.

The most basic example of which is simply this: given a pattern p, find all occurrences of p in s. The time should be O(|p| + k) where k is the number of occurrences of p in s.

An ideal solution to this problem will take O(n) time to do the preprocessing, and O(n) space to store the data structure.

Suffix trees are a solution to this problem, with all these ideal properties. They can be used to solve many other problems as well.

[In this lecture, we're going to consider the alphabet size to be O(1)]

## Tries

A trie is a data structure for storing a set of strings. Each edge of the tree is labeled with a character of the alphabet. Each node then implicitly represents a certain string of characters. Specifically a node N represents the string of letters on the edges that we follow to get from the root to N. Each node has a bit in it that indicates whether the path from the root to this node is a member of the set.

Since our alphabet is small, we can use an array of pointers at each node to point at the subtrees of it. So to determine if a pattern p occurs in our set we simply traverse down from the root of the tree one character at a time until we either (1) walk off the bottom of the tree, in which case p does not occur, or (2) we stop at some node M. If M is marked, then p is in our set, otherwise it is not.

This process takes O(|p|) time because each step simply looks up the next character of p in an array of child pointers from the current node.

Note that if we were to keep a count at each node of the number of marked nodes in the subtree rooted there, we could then efficiently

determine for a pattern p, how many members of my set of strings begin with the characters of p.

## Now, returning to suffix trees

Our first attempt to build a data structure that solves this problem is to build a trie which stores all the strings which are suffixes of the given string s. It's going to be useful to avoid having one suffix match the beginning of another suffix. So in order to avoid this we will affix a special character denoted "$" at the end of the string s, which occurs nowhere else in s. (This character is lexicographically less than any other character.)

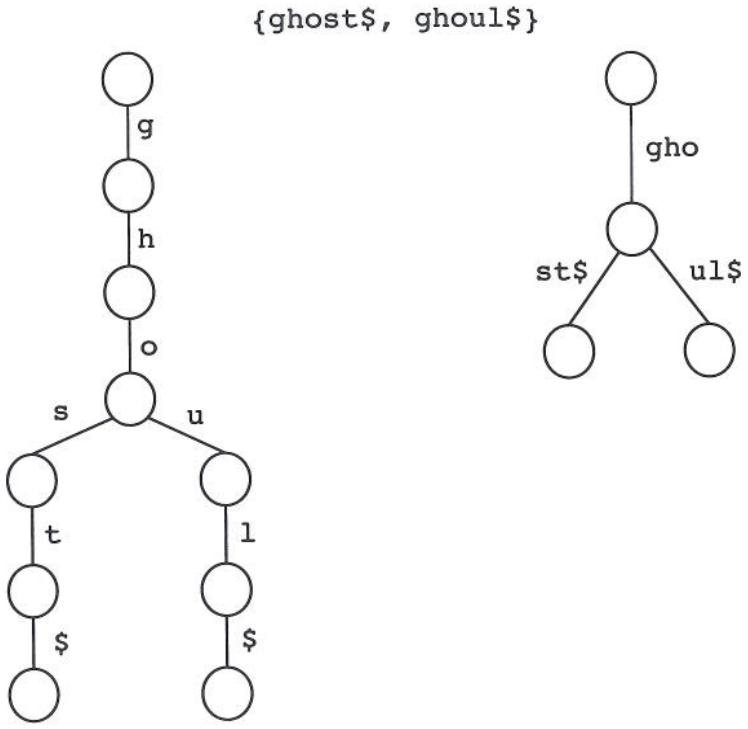
Suppose we build a trie as described above using all the suffixes of s, and we added the counts as described above to the trie.

Now given a pattern p, we can count the number of occurrences of p in s in O(|p|) time. We just walk down the trie and when we run out of p we look at the count of the node we're sitting on. It's our answer.

But there are a number of problems with this solution. First of all, the space to store this data structure could be as large as O(n^2). And it will also take too long to build it. Also, it's unsatisfactory in that it does not tell us where in s these patterns occur.

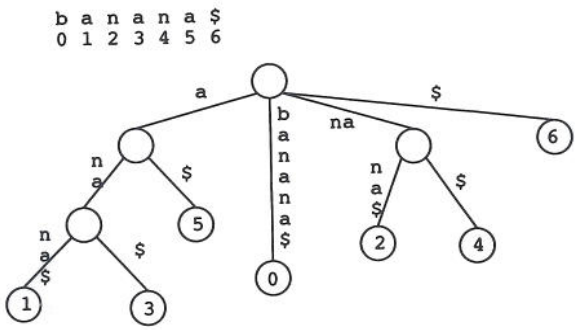
Because no string occurs as a prefix of any other, we can divide the nodes of our trie into internal and leaf nodes. The leaf nodes have no children, and represent a suffix of s. So we can have the leaf node point to the place in s where the given suffix begins.

We can also get the space consumption down to O(n). Suppose in the trie there is a long path with branching factor 1 at each node on that path. That string of characters must occur in s, so we can represent it implicitly by a pair of pointers into the string s. So an edge is now labeled with a pair of indices into s instead of just a single character.

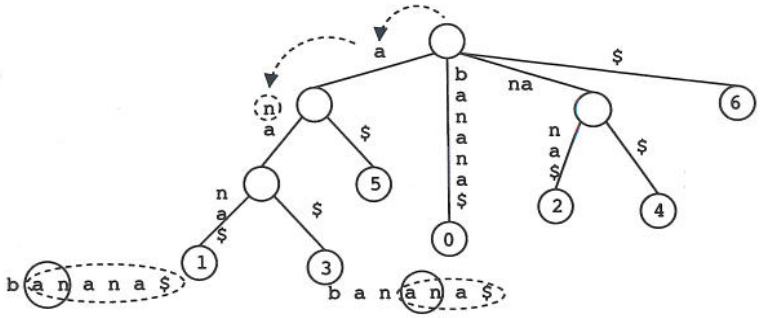



Uncompressed and compressed tries for *ghost* and *ghoul*

This representation uses O(n) space. (We count pointers as O(1) space.) One nice way to see this is to imagine building this data structure by adding suffixes into it one at a time. To add a new suffix, we walk down the current tree until we come to a place where the path leads off of the current tree. (This must occur because the suffix is not already in the tree.) This could happen in the middle of an edge, or at an already existing node. In the former case, we split the edge in two and add a new node with a branching factor of 2 in the middle of it. In the latter case we simply add a new edge from an already existing node. In either case the process terminates. The number of nodes in the tree is thus O(n).




Suffix tree for *banana*



Lookup for *an* in suffix tree for *banana*

The running time of this naive construction algorithm is still O(n^2). We'll talk more later about how to make this more efficient.

## Other Applications of Suffix Trees

There are many other applications of suffix trees to practical problems on strings. Gusfield discusses many of these in his book. I'll just mention a couple here.

### 1. Longest Common Substring of Two Strings

Given two strings s and t, what is the longest substring that occurs in both of them. For example if a="boogie" and b="ogre" then the answer is "og". The question is how to compute this efficiently. And the answer is to use suffix trees. Here's how.

Construct a new string s = a%b. That is, concatenate a and b together with an intervening special character that occurs nowhere else (indicated here by "%"). Now construct the suffix tree for s. Every leaf of the suffix tree represents a suffix that begins in a or in b. Mark every internal node with two bits: one that indicates that this subtree contains a substring of a, and another for b. These marks can be computed by depth first search (linear time). Now take the deepest (in the sense of the longest string path length in the suffix tree) node in the suffix tree that has both marks. This tells you the the longest common substring.

It was believed by Knuth prior to suffix trees that this problem could not be solved in linear time.

### 2. Finding all Maximal Palindromes

A maximal palindrome in a string s is a palindrome which cannot be

extended by grabbing another character of s on each end.
Gusfield explains on pages 197 and 198 how to find all of these in O(n) time.

## Computing the Suffix Tree

I'll explain how to compute the suffix tree from two other constructs of the string s. They are the suffix array and the prefix length array.

Imagine that you write down all the suffixes of a string s. The ith suffix is the one that begins at position i. Now imagine that you sort all of these suffixes. And you write down the indices of them in an array in their sorted order. This is the suffix array.

Example: s=banana$

```
  0  1  2  3  4  5  6
  b  a  n  a  n  a  $
```

```
6:  $
5:  a$
3:  ana$
1:  anana$
0:  banana$
4:  na$
2:  nana$
```

So the suffix array is: 6 5 3 1 0 4 2

Each successive suffix in this order matches the previous one in some number of letters. This is called the common prefix lengths array. In this case we have:

```
suffix array is:                6 5 3 1 0 4 2
common prefix lengths array      0 1 3 0 0 2
```

Given these two things, the suffix tree can be computed in linear time.

We add the suffixes one at a time into a partially built suffix tree in the order that they appear in the suffix array. We keep at any point in time the sequence of nodes on the path from the most recently added leaf to the root. To add the next suffix, we find where its path deviates from the current one. To do this we use the common prefix length value. We walk up the path until we pass this prefix length. This tells us where to add the new node.

The time to build the suffix tree in this fashion is the same as the time it takes to traverse the suffix tree from left to right. That is, it's linear time.

So how do we compute the suffix array and the common prefix lengths array? There are linear time algorithms for this, but here I will describe a probabilistic method that is O(n log^2 n).

It's based on Karp-Rabin fingerprinting. If we could compare two suffixes in O(1) time we could then just sort them in O(n log n) time. Instead use a method for comparing two suffixes that works in O(log n) time.

Using Karp-Rabin fingerprinting we can in O(1) time (as I explained in the last lecture) compare two substrings for equality. To compare two suffixes for lexicographic order, we use binary search to find the

shortest length R such that the first R characters of each of the suffixes differ, but the first R-1 characters of them are the same. Then the lexicographic order is determined by the Rth character of them. Furthermore this also tells us the common prefix length between the two strings.

Here's a java implementation of this technique.

```
/*
   O(n log^2(n) algorithm to compute the suffix array of a string based on
   Karp-Rabin fingerprinting.

   D. Sleator   Dec 4, 2012

*/
import java.io.*;
import java.util.*;

public class Suffix_Array {
    static final long P = 1000000007;
    static long[] p;  // p[i] = P^i modulo 2^64
    static long[] a;  // a[i] = s[i-1]*p[0] + s[i-2]*p[1] + ... + s[0]*p[i-1]
    static char[] s;
    static int n;

    static long hh(int x, int y) {
        /* Assumes x<=y.  Let k = y-x.  This function returns
         * s[x]*p[0] + s[x+1]*p[1] + ... + s[y]*p[k].
         * In other words, it's the hash function from x to y inclusive
         */
        return a[y+1]-a[x]*p[y-x+1];
    }

    static int pre_len;  /* a side effect of comp, which is the common prefix
                            length of the two strings just compared */

    static int comp (int x, int y) {
        /* Compare the two strings which are the suffix of s beginning
         * at x and beginning at y.  Return <0, 0, or >0 depending
         * on the outcome. (Actually in this context they can't be equal.)
         */
        int R = Math.min(n-x-1,n-y-1);
        int L=0;
        while(L<R) {
            /* Loop invariant:
             * these two strings are equal: x[0..L-1], y[0..L-1]
             * these two strings are not equal x[0..R], y[0..R]
             */
            int M=(L+R+1)/2;
            if (hh(x,x+M-1) == hh(y,y+M-1)) L=M; else R=M-1;
        }
        pre_len = R;
        return s[x+R] - s[y+R];
    }

    public static void main(String[] args) {
        if (args.length <= 0) {
            System.out.printf("Supply a string\n");
            System.exit(1);
        }
        String input = args[0] + "\0";
        s = input.toCharArray();
        n = s.length;

        p = new long[n+1];
        a = new long[n+1];
```

```
        /* precompute p[] and a[] to make hh() work in O(1) time */
        p[0]=1;
        for(int i=1; i<=n; i++) p[i] = p[i-1] * P;
        for(int i=1; i<=n; i++) a[i]=a[i-1]*P+s[i-1];

        Integer[] perm = new Integer[n];
        for (int i=0; i<n; i++) perm[i] = i;

        Arrays.sort(perm, new Comparator<Integer>() {
                public int compare(Integer A, Integer B) {return comp(A,B);}
            });

        int[] prefix = new int[n-1];
        for (int i=0; i<n-1; i++) {
            comp(perm[i],perm[i+1]);
            prefix[i] = pre_len;
        }

        System.out.printf("Suffix Array: ");
        for(int i=0; i<n; i++) {
            System.out.printf("%d ", perm[i]);
        }
        System.out.println();

        System.out.printf("Common Prefix Lengths: ");
        for(int i=0; i<n-1; i++) {
            System.out.printf("%d ", prefix[i]);
        }
        System.out.println();

    }
}
```

Sample Runs:

```
$ java Suffix_Array "banana"
Suffix Array: 6 5 3 1 0 4 2
Common Prefix Lengths: 0 1 3 0 0 2

$ java Suffix_Array "mississippi"
Suffix Array: 11 10 7 4 1 0 9 8 6 3 5 2
Common Prefix Lengths: 0 1 1 4 0 0 1 0 2 1 3

$ java Suffix_Array "1111000011110000"
Suffix Array: 16 15 14 13 12 4 5 6 7 11 3 10 2 9 1 8 0
Common Prefix Lengths: 0 1 2 3 4 3 2 1 0 5 1 6 2 7 3 8
```