CSE 549: Suffix Tries & Suffix Trees

All slides in this lecture not marked with “*” of Ben Langmead.
KMP is great, but

$$|T| = m \quad |P| = n$$ (note: m,n are opposite from previous lecture)

<table>
<thead>
<tr>
<th>Find an occurrence of P</th>
<th>With preprocessing (KMP)</th>
<th>Given preprocessing (KMP)</th>
<th>With preprocessing (ST)</th>
<th>Given preprocessing (ST)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$O(m+n)$</td>
<td>$O(m)$</td>
<td>$O(m+n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Find all occurrences of P</th>
<th>With preprocessing (KMP)</th>
<th>Given preprocessing (KMP)</th>
<th>With preprocessing (ST)</th>
<th>Given preprocessing (ST)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$O(m+n)$</td>
<td>$O(m)$</td>
<td>$O(m + n + k)$</td>
<td>$O(n+k)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Find an occurrence of P</th>
<th>With preprocessing (KMP)</th>
<th>Given preprocessing (KMP)</th>
<th>With preprocessing (ST)</th>
<th>Given preprocessing (ST)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$O(\ell(m+n))$</td>
<td>$O(\ell(m))$</td>
<td>$O( m + \ell n)$</td>
<td>$O(\ell n)$</td>
</tr>
</tbody>
</table>

If the text is constant over many patterns, pre-processing the text rather than the pattern is better (and allows other efficient queries).
Tries

A trie (pronounced “try”) is a rooted tree representing a collection of strings with one node per common prefix.

Smallest tree such that:

- Each edge is labeled with a character $c \in \Sigma$
- A node has at most one outgoing edge labeled $c$, for $c \in \Sigma$
- Each key is “spelled out” along some path starting at the root

Natural way to represent either a set or a map where keys are strings

This structure is also known as a $\Sigma$-tree
Tries: example

Represent this map with a trie:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>instant</td>
<td>1</td>
</tr>
<tr>
<td>internal</td>
<td>2</td>
</tr>
<tr>
<td>internet</td>
<td>3</td>
</tr>
</tbody>
</table>

The smallest tree such that:

- Each edge is labeled with a character $c \in \Sigma$
- A node has at most one outgoing edge labeled $c$, for $c \in \Sigma$
- Each key is “spelled out” along some path starting at the root
Tries: example

Checking for presence of a key $P$, where $n = |P|$, is $O(n)$ time.

If total length of all keys is $N$, trie has $O(N)$ nodes.

What about $|\Sigma|$?

Depends how we represent outgoing edges. If we don’t assume $|\Sigma|$ is a small constant, it shows up in one or both bounds.
Tries: another example

We can index $T$ with a trie. The trie maps substrings to offsets where they occur.

<table>
<thead>
<tr>
<th>ac</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ag</td>
<td>8</td>
</tr>
<tr>
<td>at</td>
<td>14</td>
</tr>
<tr>
<td>cc</td>
<td>12</td>
</tr>
<tr>
<td>cc</td>
<td>2</td>
</tr>
<tr>
<td>ct</td>
<td>6</td>
</tr>
<tr>
<td>gt</td>
<td>18</td>
</tr>
<tr>
<td>gt</td>
<td>0</td>
</tr>
<tr>
<td>ta</td>
<td>10</td>
</tr>
<tr>
<td>tt</td>
<td>16</td>
</tr>
</tbody>
</table>
class TrieMap(object):
    """ Trie implementation of a map. Associating keys (strings or other sequence type) with values. Values can be any type. """

    def __init__(self, kvs):
        self.root = {}
        # For each key (string)/value pair
        for (k, v) in kvs: self.add(k, v)

    def add(self, k, v):
        """ Add a key-value pair """
        cur = self.root
        for c in k: # for each character in the string
            if c not in cur:
                cur[c] = {} # if not there, make new edge on character c
            cur = cur[c]
        cur['value'] = v # at the end of the path, add the value

    def query(self, k):
        """ Given key, return associated value or None """
        cur = self.root
        for c in k:
            if c not in cur:
                return None # key wasn't in the trie
            cur = cur[c]
        # get value, or None if there's no value associated with this node
        return cur.get('value')

Python example:
http://nbviewer.ipython.org/6603619
Indexing with suffixes

Some indices (e.g. the inverted index) are based on extracting substrings from $T$

A very different approach is to extract suffixes from $T$. This will lead us to some interesting and practical index data structures:

- Suffix Trie
- Suffix Tree
- Suffix Array
- FM Index
Trie Definitions

A $\Sigma$-tree (trie) is a rooted tree where each edge is labeled with a single character $c \in \Sigma$, such that no node has two outgoing edges labeled with the same character.

- for a node $v$ in $T$, $\text{depth}(v)$ or $\text{node-depth}(v)$ is the distance from $v$ to the root.

- $\text{node-depth}(r) = 0$

- $\text{string}(v) = \text{concatenation of all characters on the path } r \leadsto v$

- $\text{string-depth}(v) = |\text{string}(v)|$ (note: $\text{string-depth}(v) \geq \text{node-depth}(v)$)

- for a string $x$, if $\exists$ node $v$ with $\text{string}(v) = x$, we say $\text{node}(x) = v$

- $T$ displays string $x$ if $\exists$ node $v$ and string $y$ such that $xy = \text{string}(v)$

- $\text{words}(T) = \{ x | T \text{ displays } x \}$

- A suffix trie of string $s$ is a $\Sigma$-tree such that $\text{words}(T) = \{s' | s' \text{ is a substring of } s\}$

- An internal/leaf edge leads to an internal/leaf node

Defs. from: http://profs.sci.univr.it/~liptak/ALBioinfo/files/sequence_analysis.pdf
Suffix trie

Build a trie containing all suffixes of a text $T$
Suffix trie

First add special *terminal character* $\$ to the end of $T$

$\$ is a character that does not appear elsewhere in $T$, and we define it to be less than other characters (for DNA: $\$ < A < C < G < T$)

$\$ enforces a rule we’re all used to using: e.g. “as” comes before “ash” in the dictionary. $\$ also guarantees no suffix is a prefix of any other suffix.
Suffix trie

$T$: abaaba \quad T\$: abaaba$\$

Each path from root to leaf represents a suffix; each suffix is represented by some path from root to leaf.

Would this still be the case if we hadn’t added $\$$?
Suffix trie

T: abaaba

Each path from root to leaf represents a suffix; each suffix is represented by some path from root to leaf

Would this still be the case if we hadn’t added $?  No
Suffix trie

We can think of nodes as having **labels**, where the label spells out characters on the path from the root to the node.
Suffix trie

How do we check whether a string $S$ is a substring of $T$?

Note: Each of $T$'s substrings is spelled out along a path from the root. I.e., every substring is a prefix of some suffix of $T$.

Start at the root and follow the edges labeled with the characters of $S$

If we “fall off” the trie -- i.e. there is no outgoing edge for next character of $S$, then $S$ is not a substring of $T$

If we exhaust $S$ without falling off, $S$ is a substring of $T$
**Suffix trie**

How do we check whether a string $S$ is a substring of $T$?

Note: Each of $T$'s substrings is spelled out along a path from the root. I.e., every *substring* is a *prefix* of some *suffix* of $T$.

Start at the root and follow the edges labeled with the characters of $S$

- If we “fall off” the trie -- i.e. there is no outgoing edge for next character of $S$, then $S$ is not a substring of $T$
- If we exhaust $S$ without falling off, $S$ is a substring of $T$

$S = \text{abaaba}$

Yes, it’s a substring
Suffix trie

How do we check whether a string $S$ is a substring of $T$?

Note: Each of $T$'s substrings is spelled out along a path from the root. I.e., every substring is a prefix of some suffix of $T$.

Start at the root and follow the edges labeled with the characters of $S$

If we “fall off” the trie -- i.e. there is no outgoing edge for next character of $S$, then $S$ is not a substring of $T$

If we exhaust $S$ without falling off, $S$ is a substring of $T$
Suffix trie

How do we check whether a string $S$ is a suffix of $T$?

Same procedure as for substring, but additionally check whether the final node in the walk has an outgoing edge labeled $\$$. 

$S = baa$

Not a suffix
Suffix trie

How do we check whether a string $S$ is a suffix of $T$?

Same procedure as for substring, but additionally check whether the final node in the walk has an outgoing edge labeled $\$$.
Suffix trie

How do we count the number of times a string $S$ occurs as a substring of $T$?

Follow path corresponding to $S$. Either we fall off, in which case answer is 0, or we end up at node $n$ and the answer = # of leaf nodes in the subtree rooted at $n$.

Leaves can be counted with depth-first traversal.
Suffix trie

How do we find the **longest repeated substring** of $T$?

Find the deepest node with more than one child
Suffix trie: implementation

class SuffixTrie(object):

def __init__(self, t):
    ''' Make suffix trie from t '''
    t += '$' # special terminator symbol
    self.root = {}
    for i in xrange(len(t)): # for each suffix
        cur = self.root
        for c in t[i:]: # for each character in i'th suffix
            if c not in cur:
                cur[c] = {} # add outgoing edge if necessary
            cur = cur[c]

def followPath(self, s):
    ''' Follow path given by characters of s. Return node at end of path, or None if we fall off. '''
    cur = self.root
    for c in s:
        if c not in cur:
            return None
        cur = cur[c]
    return cur

def hasSubstring(self, s):
    ''' Return true iff s appears as a substring of t '''
    return self.followPath(s) is not None

def hasSuffix(self, s):
    ''' Return true iff s is a suffix of t '''
    node = self.followPath(s)
    return node is not None and '$' in node

Python example:  http://nbviewer.ipython.org/6603756
Suffix trie

How many nodes does the suffix trie have?

Is there a class of string where the number of suffix trie nodes grows linearly with $m$?

Yes: e.g. a string of $m$ a’s in a row ($a^m$)

$T = aaaa$

1 Root

- $m$ nodes with incoming $a$ edge
- $m + 1$ nodes with incoming $\$ edge

$2m + 2$ nodes
Suffix trie

Is there a class of string where the number of suffix trie nodes grows with $m^2$?

Yes: $a^n b^n$

- 1 root
- $n$ nodes along “b chain,” right
- $n$ nodes along “a chain,” middle
- $n$ chains of $n$ “b” nodes hanging off each “a chain” node
- $2n + 1$ $\$ leaves (not shown)

$n^2 + 4n + 2$ nodes, where $m = 2n$
Could worst-case # nodes be worse than $O(m^2)$?

Max # nodes from top to bottom
= length of longest suffix + 1
= $m + 1$

Max # nodes from left to right
= max # distinct substrings of any length
$\leq m$

$O(m^2)$ is worst case
Suffix trie: actual growth

Built suffix tries for the first 500 prefixes of the lambda phage virus genome

Black curve shows how # nodes increases with prefix length
Suffix tries -> Suffix trees
Suffix Tree Definitions

A $\Sigma^+$-tree is a rooted tree, $T$, where each edge is labeled with non-empty strings, where no node has two outgoing edges labeled with strings having the same first character. $T$ is compact if all internal nodes have $\geq 2$ children.

- for a node $v$ in $T$, $\text{depth}(v)$ or $\text{node-depth}(v)$ is the distance from $v$ to the root.

- $\text{node-depth}(r) = 0$

- $\text{string}(v) =$ concatenation of all characters on the path $r \leadsto v$

- $\text{string-depth}(v) = |\text{string}(v)|$ (note: $\text{string-depth}(v) \geq \text{node-depth}(v)$)

- for a string $x$, if $\exists$ node $v$ with $\text{string}(v) = x$, we say $\text{node}(x) = v$

- $T$ displays string $x$ if $\exists$ node $v$ and string $y$ such that $xy = \text{string}(v)$

- $\text{words}(T) = \{ x \mid T \text{ displays } x \}$

- A suffix tree of string $s$ is a compact $\Sigma^+$-tree such that $\text{words}(T) = \{s' \mid s'$ is a substring of $s\}$

Suffix trie: making it smaller

$T = \text{abaaba}\$

**Idea 1:** Coalesce non-branching paths into a *single edge* with a *string* label

Reduces # nodes, edges, guarantees internal nodes have >1 child
**Suffix tree**

$L$ leaves, $I$ internal nodes, $E$ edges

$E = L + I - 1$

$E \geq 2I$ (each internal node branches)

$L + I - 1 \geq 2I \Rightarrow I \leq L - 1$

*but*

$L \leq m$ (at most $m$ suffixes)

$I \leq m - 1$

$E = L + I - 1 \leq 2m - 2$

$E + L + I \leq 4m - 3 \in O(m)$

*Is the total size $O(m)$ now?*
Suffix tree

$L$ leaves, $I$ internal nodes, $E$ edges

\[ E = L + I - 1 \]

\[ E \geq 2I \text{ (each internal node branches)} \]

\[ L + I - 1 \geq 2I \implies I \leq L - 1 \]

*but*

\[ L \leq m \text{ (at most } m \text{ suffixes)} \]

\[ I \leq m - 1 \]

\[ E = L + I - 1 \leq 2m - 2 \]

\[ E + L + I \leq 4m - 3 \in O(m) \]

Is the total size $O(m)$ now? **No**: total length of edge labels is quadratic in $m$
Idea 2: Store $T$ itself in addition to the tree. Convert tree’s edge labels to (offset, length) pairs with respect to $T$.

Space required for suffix tree is now $O(m)$
Suffix tree: leaves hold offsets

\[ T = \text{abaaba} \$

\[ (3, 4) \]

\[ (6, 1) \]

\[ (0, 1) \]

\[ (6, 1) \]

\[ (1, 2) \]

\[ (3, 4) \]
Suffix tree: labels

$T = \text{abaaba}$

Again, each node's *label* equals the concatenated edge labels from the root to the node. These aren't stored explicitly.
Because edges can have string labels, we must distinguish two notions of "depth":

- **Node** depth: how many edges we must follow from the root to reach the node.
- **Label** depth: total length of edge labels for edges on path from root to node.
Suffix tree: space caveat

We say the space taken by the edge labels is $O(m)$, because we keep 2 integers per edge and there are $O(m)$ edges.

To store one such integer, we need enough bits to distinguish $m$ positions in $T$, i.e. $\text{ceil}(\log_2 m)$ bits. We usually ignore this factor, since 64 bits is plenty for all practical purposes.

Similar argument for the pointers / references used to distinguish tree nodes.

Minor point:
Suffix tree: building

Naive method 1: build a suffix trie, then coalesce non-branching paths and relabel edges

Naive method 2: build a single-edge tree representing only the longest suffix, then augment to include the 2\textsuperscript{nd}-longest, then augment to include 3\textsuperscript{rd}-longest, etc

Both are $O(m^2)$ time, but first uses $O(m^2)$ space while second uses $O(m)$

Naive method 2 is described in Gusfield 5.4

Python implementation at: [http://nbviewer.ipython.org/6665861](http://nbviewer.ipython.org/6665861)
WOTD (Write-Only Top-Down) Construction


Build a suffix tree for string s$

Recursive construction:

For every branching node \texttt{node}(u), subtree of \texttt{node}(u) is determined by all suffixes of s where u is a prefix.

Recursively construct subtree for all suffixes where u is a prefix.

**Definition**: remaining suffixes of u

\[ R(\text{node}(u)) = \{ v \mid uv \text{ is a suffix of } s \} \]
WOTD (Write-Only Top-Down) Construction

Build a suffix tree for string $s$

Recursive construction:

For every branching node $\text{node}(u)$, subtree of $\text{node}(u)$ is determined by all suffixes of $s$ where $u$ is a prefix.

Recursively construct subtree for all suffixes where $u$ is a prefix.

**Definition**: remaining suffixes of $u$

$$R(\text{node}(u)) = \{ v \mid uv \text{ is a suffix of } s \}$$

**Definition**: $c$-group of $\text{node}(u)$

$$\text{group}(\text{node}(u), c) = \{ w \in \Sigma^* \mid cw \in R(\text{node}(u)) \}$$
WOTD (Write-Only Top-Down) Construction

def WOTD(T : tree, node(u): node):
    for each c ∈ Σ ∪ {$}:
        G = group(node(u), c)
        ucv = lcp(G)
        if |G| == 1:
            add leaf node(ucv) as a child of node(u)
        else:
            add inner node(ucv) as a child of node(u)
            WOTD(T, node(ucv))

Start the algorithm by calling WOTD(T, node(ε))
s = ttatctctcta
s = ttatctctta

WOTD Example

s = ttatctctta

s = ttatctctta

WOTD Example

\[ s = \texttt{ttatctcttta} \]
WOTD Properties

- Worst case time still $\in O(|T|^2)$
- Expected case time $\in O(|T| \log |T|)$
- Write-only property & recursive construction lends itself well to parallelism
- Good caching properties (locality of reference for substrings belonging to a subtree)
- Top-down construction order allows lazy construction as discussed in:

Suffix tree: building

Other methods for construction:


\(O(m)\) time and space

Has *online* property: if \(T\) arrives one character at a time, algorithm efficiently updates suffix tree upon each arrival

We won’t cover it here; see Gusfield Ch. 6 for details

Or just Google “Ukkonen’s algorithm”
Suffix tree: actual growth

Built suffix trees for the first 500 prefixes of the lambda phage virus genome

Black curve shows # nodes increasing with prefix length

Compare with suffix trie:

123 K nodes

0 200 400 600 800 1000
# suffix tree nodes

0 100 200 300 400 500
Length prefix over which suffix tree was built
**Suffix tree**

How do we check whether a string $S$ is a substring of $T$?

Essentially same procedure as for suffix trie, except we have to deal with coalesced edges.
Suffix tree

How do we check whether a string $S$ is a suffix of $T$?

Essentially same procedure as for suffix trie, except we have to deal with coalesced edges

$S = aba$

Yes, it’s a suffix
Suffix tree

How do we count the **number of times** a string $S$ occurs as a substring of $T$?

Same procedure as for suffix trie

$S = \text{aba}$

Occurs twice
**Suffix tree: applications**

With suffix tree of $T$, we can find all matches of $P$ to $T$. Let $k = \# \text{ matches}$.

E.g., $P = ab$, $T = abaaba$.

Step 1: walk down $ab$ path

- If we “fall off” there are no matches

Step 2: visit all leaf nodes below

- Report each leaf offset as match offset

$O(n + k)$ time
Suffix tree application: find long common substrings

Dots are *maximal unique matches* (MUMs), a kind of long substring shared by two sequences.

**Red** = match was between like strands, **green** = different strands

Axes show different strains of *Helicobacter pylori*, a bacterium found in the stomach and associated with gastric ulcers.
To find the longest common substring (LCS) of $X$ and $Y$, make a new string $X \# Y \$ where $\#$ and $\$ $ are both terminal symbols. Build a suffix tree for $X \# Y \$ .

$X = xabxa$
$Y = babxba$
$X \# Y \$ = xabxa#babxba$

Consider leaves:
- offsets in [0, 4] are suffixes of $X$
- offsets in [6, 11] are suffixes of $Y$

Traverse the tree and annotate each node according to whether leaves below it include suffixes of $X$, $Y$ or both.

The deepest node annotated with both $X$ and $Y$ has LCS as its label. $O(|X| + |Y|)$ time and space.
Suffix tree application: generalized suffix trees

This is one example of many applications where it is useful to build a suffix tree over many strings at once

Such a tree is called a generalized suffix tree. These are introduced in Gusfield 6.4.
Longest Common Extension

Longest common extension: We are given strings S and T. In the future, many pairs (i,j) will be provided as queries, and we want to quickly find:

the longest substring of S starting at i that matches a substring of T starting at j.

Build generalized suffix tree for S and T.

Preprocess tree so that lowest common ancestors (LCA) can be found in constant time.

Create an array mapping suffix numbers to leaf nodes.

Given query (i,j):
  Find the leaf nodes for i and j
  Return string of LCA for i and j
Longest Common Extension

Longest common extension: We are given strings $S$ and $T$. In the future, many pairs $(i, j)$ will be provided as queries, and we want to quickly find:

the longest substring of $S$ starting at $i$ that matches a substring of $T$ starting at $j$.

Build generalized suffix tree for $S$ and $T$. $O(|S| + |T|)$

Preprocess tree so that lowest common ancestors (LCA) can be found in constant time. $O(|S| + |T|)$

Create an array mapping suffix numbers to leaf nodes. $O(|S| + |T|)$

Given query $(i, j)$:

- Find the leaf nodes for $i$ and $j$ $O(1)$
- Return string of LCA for $i$ and $j$ $O(1)$
Using LCE to Find Palindromes

Maximal even palindrome at position $i$: the longest string to the left and right so that the left half is equal to the reverse of the right half.

Goal: find all maximal palindromes in $S$. 

$x \neq y$
Using LCE to Find Palindromes

Maximal even palindrome at position $i$: the longest string to the left and right so that the left half is equal to the reverse of the right half.

**Goal:** find all maximal palindromes in $S$.

Construct $S^r$, the reverse of $S$.

Preprocess $S$ and $S^r$ so that LCE queries can be solved in constant time (previous slide).

LCE($i$, $n$-$i$) is the length of the longest palindrome centered at $i$.

For every position $i$:
  Compute LCE($i$, $n$-$i$)
Using LCE to Find Palindromes

Maximal even palindrome at position i: the longest string to the left and right so that the left half is equal to the reverse of the right half.

\[
\text{S} \quad \text{\textcolor{red}{x}} \quad \text{\textcolor{green}{y}} \quad \text{\textcolor{red}{x} \neq y}
\]

\[
i
\]

Goal: find all maximal palindromes in S.

\[
\text{S}^r \quad \text{\textcolor{green}{y}} \quad \text{\textcolor{red}{x}} \quad \text{\textcolor{red}{x} \neq y}
\]

\[
n - i
\]

Construct \( S^r \), the reverse of S. \( O(|S|) \)

Preprocess S and \( S^r \) so that LCE queries can be solved in constant time (previous slide). \( O(|S|) \)

LCE(i, n-i) is the length of the longest palindrome centered at i.

For every position i:

\[
\text{Compute LCE(i, n-i)} \quad O(|S|) \quad O(1) \quad \text{Total time} = O(|S|)
\]
Suffix trees in the real world: MUMmer

<table>
<thead>
<tr>
<th>Columns:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Match offset in T</td>
<td></td>
</tr>
<tr>
<td>2. Match offset in P</td>
<td></td>
</tr>
<tr>
<td>3. Length of exact match</td>
<td></td>
</tr>
</tbody>
</table>

Indexing phase: ~2 minutes

Matching phase: very fast

**FASTA file containing “reference” (“text”)**

**FASTA file containing ALU string**

```
Bens-MacBook-Pro:mummer langmead$ cat alu50.fa
>Alu
GGCGGTTGGCTACGCCTGTAATCCGACCTTTGGGAGGGCAGGCGGG
Bens-MacBook-Pro:mummer langmead$ $HOME/software/MUMmer3.23/mummer -maxmatch $HOME/fasta/hg19/chr1.fa alu50.fa
# reading input file "/Users/langmead/fasta/hg19/chr1.fa" of length 249250621
# construct suffix tree for sequence of length 249250621
# (maximum reference length is 536870908)
# (maximum query length is 4294967295)
# process 2492506 characters per dot
#...  
# CONSTRUCTIONTIME /Users/langmead/software/MUMmer3.23/mummer /Users/langmead/fasta/hg19/chr1.fa 125.30
# reading input file "alu50.fa" of length 50
# matching query-file "alu50.fa"
# against subject-file "/Users/langmead/fasta/hg19/chr1.fa"

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>61769671</td>
<td>1</td>
</tr>
<tr>
<td>219929011</td>
<td>1</td>
</tr>
<tr>
<td>162396657</td>
<td>1</td>
</tr>
<tr>
<td>109737840</td>
<td>1</td>
</tr>
<tr>
<td>82615090</td>
<td>1</td>
</tr>
<tr>
<td>32983678</td>
<td>1</td>
</tr>
<tr>
<td>84730371</td>
<td>1</td>
</tr>
<tr>
<td>248036256</td>
<td>1</td>
</tr>
<tr>
<td>150558745</td>
<td>1</td>
</tr>
<tr>
<td>11127213</td>
<td>1</td>
</tr>
<tr>
<td>236885661</td>
<td>1</td>
</tr>
<tr>
<td>31639677</td>
<td>1</td>
</tr>
<tr>
<td>16027333</td>
<td>1</td>
</tr>
<tr>
<td>21577225</td>
<td>1</td>
</tr>
<tr>
<td>26327837</td>
<td>1</td>
</tr>
<tr>
<td>243352583</td>
<td>1</td>
</tr>
</tbody>
</table>
```
Suffix trees in the real world: MUMmer

MUMmer v3.32 time and memory scaling when indexing increasingly larger fractions of human chromosome 1

For whole chromosome 1, took 2m:14s and used 3.94 GB memory
Suffix trees in the real world: MUMmer

Attempt to build index for whole human genome reference:

```
mummer: suffix tree construction failed: textlen=3101804822
larger than maximal textlen=536870908
```

We can predict it would have taken about 47 GB of memory
Suffix trees in the real world: the constant factor

While $O(m)$ is desirable, the constant in front of the $m$ limits wider use of suffix trees in practice

Constant factor varies depending on implementation:

Estimate of MUMmer’s constant factor = 3.94 GB / 250 million nt
≈ 15.75 bytes per node

Literature reports implementations achieving as little as 8.5 bytes per node, but no implementation used in practice that I know of is better than ≈ 12.5 bytes per node

Suffix tree: summary

Organizes all suffixes into an incredibly useful, flexible data structure, in $O(m)$ time and space.

A naive method (e.g. suffix trie) could easily be quadratic or worse.

Used in practice for whole genome alignment, repeat identification, etc.

Actual memory footprint (bytes per node) is quite high, limiting usefulness.

Actual memory footprint (bytes per node) is quite high, limiting usefulness.