Problem
• Given P and S – find all exact or approximate occurrences of P in S
• You are allowed to preprocess S (and P, of course)
• Goal: to speed up the searches

E.g. Dictionary problem
• Does P belong to a dictionary \( D=\{d_1, \ldots, d_n\} \)
  – Build a binary search tree of D
  – B-Tree of D
  – Hashing
  – Sorting + Binary search
• Build a keyword trie: search in \( O(|P|) \)
  – Assuming alphabet has up to a constant size c
  – See Aho-Corasick algorithm, Trie construction

Sorted array and binary search

Sorted array and binary search

Trie for \( D=\{ \text{he, hers, his, she} \} \)
S != set of words
- S of length n
- How to index?
- Index from every position of a text
- Prefix of every possible suffix is important

Suffix tree
- **Definition**: A compact representation of a trie corresponding to the suffixes of a given string where all nodes with one child are merged with their parents.
- **Definition (suffix tree)**: A suffix tree \( T \) for a string \( S \) (with \( n = |S| \)) is a rooted, labeled tree with a leaf for each non-empty suffix of \( S \). Furthermore, a suffix tree satisfies the following properties:
  - Each internal node, other than the root, has at least two children;
  - Each edge leaving a particular node is labeled with a non-empty substring of \( S \) of which the first symbol is unique among all first symbols of the edge labels of the edges leaving this particular node;
  - For any leaf in the tree, the concatenation of the edge labels on the path from the root to this leaf exactly spells out a non-empty suffix of \( S \).
- **Definition**: A suffix tree \( T \) for a string \( T \) is a compacted trie that represents all the suffixes of string \( T \).
- Linear size: \( |Tree(T)| = O(|T|) \)
- Can be constructed in linear time \( O(|T|) \)
- Has myriad virtues (A. Apostolico)
- Is well-known: 366 000 Google hits

The suffix tree Tree(\( T \)) of \( T \)
- Data structure **suffix tree**, Tree(\( T \)), is a compacted trie that represents all the suffixes of string \( T \)
- Linear size: \( |Tree(T)| = O(|T|) \)
- Can be constructed in linear time \( O(|T|) \)
- Has myriad virtues (A. Apostolico)
- Is well-known: 366 000 Google hits

Literature on suffix trees

Partly based on:

Suffix tree and suffix array techniques for pattern analysis in strings
Esko Ukkonen
Univ Helsinki
Erice School 30 Oct 2005

High-throughput genome-scale sequence analysis and mapping using compressed data structures
Veli Mäkinen
Department of Computer Science
University of Helsinki

Analysis of a string of symbols

- **T** = hattivatti  ‘text’
- **P** = att  ‘pattern’

- Find the occurrences of **P** in **T**: hattivatti
- Pattern synthesis:  
  \( # (t) = 4 \)  
  \( # (atti) = 2 \)  
  \( # (t****t) = 2 \)

Pattern finding & synthesis problems

- **T** = t1t2 ... tp, **P** = p1p2 ... pn, strings of symbols in finite alphabet
- Indexing problem: Preprocess **T** (build an index structure) such that the occurrences of different patterns **P** can be found fast
  - static text, any given pattern **P**
- Pattern synthesis problem: Learn from **T** new patterns that occur surprisingly often
- What is a pattern? Exact substring, approximate substring, with generalized symbols, with gaps, ...

Solution: backtracking with suffix tree

...ACACATATCAACCGGCAATCGGCATTACCGGTAAGTGCG...

11/27/16
The suffix tree Tree(T) of T

- data structure suffix tree. Tree(T), is compacted trie that represents all the suffixes of string T
- linear size: |Tree(T)| = O(|T|)
- can be constructed in linear time O(|T|)
- has myriad virtues (A. Apostolico)
- is well-known: 366 000 Google hits

Suffix trie and suffix tree


Trie(T) can be large

- |Trie(T)| = O(|T|^2)
- bad example: T = a^n b^n
- Trie(T) can be seen as a DFA: language accepted = the suffixes of T
- minimize the DFA => directed cyclic word graph (‘DAWG’)

Tree(T) is of linear size

- only the internal branching nodes and the leaves represented explicitly
- edges labeled by substrings of T
- v = node(α) if the path from root to v spells α
- one-to-one correspondence of leaves and suffixes
- |T| leaves, hence < |T| internal nodes
- |Tree(T)| = O(|T| + size(edge labels))

Tree(hattivatti)
Tree(hattivatti)

substring labels of
edges represented as
pairs of pointers

Tree(T) is full text index

P occurs in T at
locations 8, 31, ...
P occurs in T  P is a prefix of some suffix of T
Path for P exists in Tree(T)
All occurrences of P in time $O(|P| + \#occ)$

Find att from Tree(hattivatti)

Linear time construction of Tree(T)

- $T = t_1t_2...t_n$ $\$ 
- $P_i = t_1t_2...t_i$ $i$th prefix of $T$
- on-line idea: update $Trie(P_i)$ to $Trie(P_{i+1})$
- => very simple construction
Trie(abaab)

chain of links connects the end points of current suffixes

Trie(abaab)

Add next symbol = b

What happens in Trie(P_i) => Trie(P_{i+1})?
What happens in \( \text{Trie}(P_i) \Rightarrow \text{Trie}(P_{i+1}) \)?

- time: \( O(\text{size of Trie}(T)) \)
- suffix links:
  \( \text{slink}(\text{node}(aa)) = \text{node}(a) \)

On-line procedure for suffix trie

1. Create Trie\((t_0)\): nodes root and \( v \), an arc son(root, \( t_0 \)) \( = v \), and suffix links \( \text{slink}(v) = \text{root} \) and \( \text{slink}(\text{root}) = \text{root} \)
2. for \( i := 2 \) to \( n \) do begin
3. \( v_i := \text{leaf of Trie}(t_i, \ldots, t_n) \) \( \text{for string } t_i, \ldots, t_n \) \( \text{(i.e., the deepest leaf)} \)
4. \( v := v_i, v' := 0 \)
5. while node \( v \) has no outgoing arc for \( t_i \) do begin
6. Create a new node \( v' \) and an arc son\((v, t_i) \) \( = v' \)
7. if \( v' \neq 0 \) then \( \text{slink}(v) := v' \)
8. \( v := \text{slink}(v); v' := v' \) end
9. for the node \( v' \) such that \( v' = \text{son}(v) \) do
  if \( v' = v \) then \( \text{slink}(v) := \text{root} \) else \( \text{slink}(v') := v' \)

Suffix trees on-line

- 'compacted version' of the on-line trie construction: simulate the construction on the linear size tree instead of the trie => time \( O(|T|) \)
- all trie nodes are conceptually still needed => implicit and real nodes

Implicit and real nodes

- Pair \((v, \alpha)\) is an implicit node in Trie\((T)\) if \( v \) is a node of Tree and \( \alpha \) is a (proper) prefix of the label of some arc from \( v \). If \( \alpha \) is the empty string then \((v, \alpha)\) is a 'real' node \( (= v) \).
- Let \( v = \text{node}(\alpha') \) in Tree\((T)\). Then implicit node \((v, \alpha)\) represents node\((\alpha' \alpha)\) of Trie\((T)\)

Implicit node

Suffix links and open arcs

- Label \([i,*]\) instead of \([i,j]\) if \( w \) is a leaf and \( j \) is the scanned position of \( T \)
Big picture

Suffix link path traversed: total work $O(n)$

new arcs and nodes created: total work $O(\text{size}(\text{Tree}(T)))$

On-line procedure for suffix tree

Input: string $T = t_1, \ldots, t_n$
Output: Tree(T)

Notation: $\text{son}(v, a) = w$ if there is an arc from $v$ to $w$ with label $a$

Function $\text{Canonize}(v, a)$:

- while $\text{son}(v, a') \neq 0$ where $a = a' a''$, $|a'| > 0$

  - $v := \text{son}(v, a')$
  - $a := a''$

- return $(v, a)$

The actual time and space

- $|\text{Tree}(T)|$ is about $20|T|$ in practice
- brute-force construction is $O(|T| \log |T|)$ for random strings as the average depth of internal nodes is $O(\log |T|)$
- difference between linear and brute-force constructions not necessarily large (Giegerich & Kurtz)
- truncated suffix trees: $k$ symbols long prefix of each suffix represented (Na et al. 2003)
- alphabet independent linear time (Farach 1997)
Applications of Suffix Trees


- **APL1**: Exact String Matching Search for P from text S. Solution 1: build STree(S) - one achieves the same $O(n+m)$ as Knuth-Morris-Pratt, for example!
- Search from the suffix tree is $O(|P|)$
- **APL2**: Exact set matching Search for a set of patterns P
Back to backtracking

Applications of Suffix Trees

• APL3: substring problem for a database of patterns
  Given a set of strings $S = S_1, \ldots, S_n$ - a database Find all $S_i$ that have $P$ as a substring
  • Generalized suffix tree contains all suffixes of all $S_i$
  • Query in time $O(|P|)$, and can identify the LONGEST common prefix of $P$ in all $S_i$

Applications of Suffix Trees

• APL4: Longest common substring of two strings
  • Find the longest common substring of $S$ and $T$
  • Overall there are potentially $O(n^2)$ such substrings, if $n$ is the length of a shorter of $S$ and $T$
  • Donald Knuth once (1970) conjectured that linear-time algorithm is impossible.
  • Solution: construct the Stree($S+T$) and find the node deepest in the tree that has suffixes from both $S$ and $T$ in subtree leaves.
  • Ex: $S =$ superiorcalifornialives $T =$ sealiver have both a substring alive.

Simple analysis task: LCSS

• Let $LCSS(A,B)$ denote the longest common substring two sequences $A$ and $B$. E.g.:
  $LCSS(AGATCTATCT, CGCCTCTAGT) = TCTAT$. 
  • A good solution is to build suffix tree for the shorter sequence and make a descending suffix walk with the other sequence.

Suffix link

Descending suffix walk

Read B left-to-right, always going down in the tree when possible.
If the next symbol of B does not match any edge label on current position, take suffix link, and try again.
(Suffix link in the root to itself emits a symbol).
The node $v$ encountered with largest string depth is the solution.
Applications of Suffix Trees

- **APL5**: Recognizing DNA contamination. Related to DNA sequencing, search for longest strings (longer than threshold) that are present in the DB of sequences of other genomes.
- **APL6**: Common substrings of more than two strings. Generalization of APL4, can be done in linear (in total length of all strings) time.

Another common tool: Generalized suffix tree

Generalized suffix tree application

Case study continued

Applications of Suffix Trees

- **APL7**: Building a directed graph for exact matching: **Suffix graph** - directed acyclic word graph (DAWG), a **smallest finite state automaton** recognizing all suffixes of a string S. This automaton can recognize membership, but not tell which suffix was matched.
- Construction: merge isomorphic subtrees.
- Isomorphic in Suffix Tree when exists suffix link path, and subtrees have equal nr. of leaves.

Applications of Suffix Trees

- **APL8**: A reverse role for suffix trees, and major space reduction. Index the pattern, not tree...
- Matching statistics.
- **APL10**: All-pairs suffix-prefix matching. For all pairs \( S_i, S_j \), find the longest matching suffix-prefix pair. Used in shortest common superstring generation (e.g. DNA sequence assembly), EST alignment etc.
Applications of Suffix Trees

- **APL11:** Finding all maximal repetitive structures in linear time
- **APL12:** Circular string linearization e.g. circular chemical molecules in the database, one wants to linearize them in a canonical way...
- **APL13:** Suffix arrays - more space reduction will touch that separately

Applications of Suffix Trees

- **APL14:** Suffix trees in genome-scale projects
- **APL15:** A Boyer-Moore approach to exact set matching
- **APL16:** Ziv-Lempel data compression
- **APL17:** Minimum length encoding of DNA

Applications of Suffix Trees

- Additional applications Mostly exercises...
  - Extra feature: **CONSTANT time lowest common ancestor retrieval (LCA)**
  - **APL:** Finding all maximal palindromes in linear time
  - **APL:** Longest common extension: a bridge to inexact matching
  - **APL:** Finding all maximal palindromes in linear time
    - Palindrome reads from central position the same to left and right. E.g.: kirik, saippuakivikauppias.
    - Build the suffix tree of $S$ and inverted $S$ (aabcbad => aabcbad#dabcbaa ) and using the LCA one can ask for any position pair ($i$, $2i-1$), the longest common prefix in constant time.
    - The whole problem can be solved in $O(n)$.

Properties of suffix tree

- **Properties of suffix tree... in practice**
  - **APL:** Exact matching with wild cards
  - **APL:** The $k$-mismatch problem
  - Approximate palindromes and repeats
  - Faster methods for tandem repeats
  - A linear-time solution to the multiple common substring problem
  - And many-many more ...

Properties of suffix tree

- **Properties of suffix tree**
  - **Suffix tree has $n$ leaves and at most $n-1$ internal nodes, where $n$ is the total length of all sequences indexed.**
  - Each node requires constant number of integers (pointers to first child, sibling, parent, text range of incoming edge, statistics counters, etc.).
  - Can be constructed in linear time.

- **Human genome itself takes less than 1 GB using 2-bits per bp.**
1. Suffix tree
2. Suffix array
3. Some applications
4. Finding motifs
Suffix array

- Many algorithms on suffix tree can be simulated using suffix array...
- ... and couple of additional arrays...
- ... forming so-called enhanced suffix array...
- ... leading to the similar space requirement as careful implementation of suffix tree
- Not a satisfactory solution to the space issue.

Pattern search from suffix array

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>hattivatti</td>
<td>11</td>
</tr>
<tr>
<td>attivatti</td>
<td>7</td>
</tr>
<tr>
<td>tlvivatti</td>
<td>2</td>
</tr>
<tr>
<td>tlvivatti</td>
<td>1</td>
</tr>
<tr>
<td>lvivatti</td>
<td>10</td>
</tr>
<tr>
<td>vivatti</td>
<td>5</td>
</tr>
<tr>
<td>atti</td>
<td>9</td>
</tr>
<tr>
<td>tti</td>
<td>4</td>
</tr>
<tr>
<td>ti</td>
<td>8</td>
</tr>
<tr>
<td>i</td>
<td>3</td>
</tr>
<tr>
<td>x</td>
<td>6</td>
</tr>
</tbody>
</table>

What we learn today?

- We learn that it is possible to replace suffix trees with compressed suffix trees that take 8.8 GB for the human genome.
- We learn that backtracking can be done using compressed suffix arrays requiring only 2.1 GB for the human genome.
- We learn that discovering interesting motif seeds from the human genome takes 40 hours and requires 9.3 GB space.

Kärkkäinen-Sanders algorithm

1. Construct the suffix array of the suffixes starting at positions i mod 3 ≠ 0. This is done by reduction to the suffix array construction of a string of two thirds the length, which is solved recursively.
2. Construct the suffix array of the remaining suffixes using the result of the first step.
3. Merge the two suffix arrays into one.

Recent suffix array constructions

- Manber&Myers (1990): $O(|T| \log |T|)$
- linear time via suffix tree
- January / June 2003: direct linear time construction of suffix array
  - Kim, Sim, Park, Park (CPM03)
  - Kärkkäinen & Sanders (ICALP03)
  - Ko & Aluru (CPM03)

Notation

- string $T = T[0..n] = t_0 t_1 ... t_{n-1}$
- suffix $S_i = T[i..0] = t_i t_{i+1} ... t_{n-1}$
- for $C \subseteq [0,n]$: $S_C = \{S_i \mid i \in C\}$

- suffix array $SA[0..n]$ of $T$ is a permutation of $[0,n]$ satisfying $S_{SA[0]} < S_{SA[1]} < ... < S_{SA[n]}$
Running example

\[ T[0,n) = y a b a d a b b a d o 0 0 \ldots \]

\[ S_A = (12,1,6,4,9,3,8,2,7,5,10,11,0) \]

Step 0: Construct a sample

- for \( k = 0,1,2 \)
  \[ B_k = \{ i \in [0,n] \mid i \mod 3 = k \} \]
- \( C = B_1 \cup B_2 \) sample positions
- \( S_C \) sample suffixes

\[ \text{Example: } B_1 = \{1,4,7,10\}, B_2 = \{2,5,8,11\}, C = \{1,4,7,10,2,5,8,11\} \]

Step 1: Sort sample suffixes

- for \( k = 1,2 \), construct
  \[ R_k = [t_1s_{k+1};t_2s_{k+2}] \ldots [t_{\text{max}}s_{k+1};t_{\text{max}}s_{k+2}] \]
  \[ R = R_1 \uplus R_2 \] concatenation of \( R_1 \) and \( R_2 \)

Suffixes of \( R \) correspond to \( S_C \); suffix \([t_i,t_{i+1}]\ldots\) corresponds to \( S_i \); correspondence is order preserving.

Sort the suffixes of \( R \): radix sort the characters and rename with ranks to obtain \( R' \). If all characters different, their order directly gives the order of suffixes. Otherwise, sort the suffixes of \( R' \) using Kärkkäinen-Sanders. Note: \( |R'| = 2n/3 \).

Step 1 (cont.)

- once the sample suffixes are sorted, assign a rank to each:
  \( \text{rank}(S_i) = \text{the rank of } S_i \text{ in } S_C \)
  \( \text{rank}(S_{n+1}) = 0 \)

- Example:
  \[ R = [abb][ada][bba][do0][bba][dab][bad][o00] \]
  \[ R' = (1,2,4,6,4,5,3,7) \]
  \( S_{A'} = (8,0,1,6,4,2,5,3,7) \)
  \( \text{rank}(S_i) \sim 0 \sim 4 \sim 2 \sim 6 \sim 5 \sim 3 \sim 7 \sim 8 \sim 0 \)

Step 2: Sort nonsample suffixes

- for each non-sample \( S_i \in S_{B_0} \) (note that rank\( (S_i) \) is always defined for \( i \in B_0 \))
  \[ S \leq S_i \iff (t_i,\text{rank}(S_{i+1})) \leq (t_j,\text{rank}(S_{j+1})) \]
- radix sort the pairs \((t_i,\text{rank}(S_{i+1}))\).

- Example: \( S_{12} \leq S_6 \leq S_8 \leq S_3 \leq S_0 \) because \( (0,0) < (a,5) < (a,7) < (b,2) < (y,1) \)

Step 3: Merge

- merge the two sorted sets of suffixes using a standard comparison-based merging;
- to compare \( S_i \in S_C \) with \( S_j \in S_{B_0} \), distinguish two cases:
  \[ i \in B_1 \mid S_i \leq S_j \iff (t_i,\text{rank}(S_{i+1})) \leq (t_j,\text{rank}(S_{j+1})) \]
  \[ i \in B_2 \mid S_i \leq S_j \iff (t_i,t_{i+1},\text{rank}(S_{i+2})) \leq (t_j,t_{j+1},\text{rank}(S_{j+2})) \]
- note that the ranks are defined in all cases!
  \[ S_1 \leq S_6 \text{ as } (a,4) < (a,5) \text{ and } S_3 \leq S_8 \text{ as } (b,a,6) < (b,a,7) \]
Running time $O(n)$

- excluding the recursive call, everything can be done in linear time
- the recursion is on a string of length $2n/3$
- thus the time is given by recurrence
  \[ T(n) = T(2n/3) + O(n) \]
- hence $T(n) = O(n)$

Implementation

- about 50 lines of C++
- code available e.g. via Juha Kärkkäinen’s home page

LCP table

- Longest Common Prefix of successive elements of suffix array:
- $LCP[i] = \text{length of the longest common prefix of suffixes } S_{SA[i]} \text{ and } S_{SA[i+1]}$
- build inverse array $SA^{-1}$ from $SA$ in linear time
- then LCP table from $SA^{-1}$ in linear time (Kasai et al, CPM2001)

- Example - Word of the Day, Fourth
- PAT index - by Gaston Gonnet (ta on samast Maple tarkvara üks looja, puudutades molekulaabiologia tarkvara paketi kasutajatele)
- PAT index is essentially a suffix array. To save space, indexed only from first character of every word
- XML-tagging (or SGML, at that time!) also indexed
- To mark certain fields of XML, the bit vectors were used.
- Main concern - improve the speed of search on the CD - minimize random accesses.
- For slow medium even 15-20 accesses is too slow...

Suffix tree vs suffix array

- suffix tree $\leftrightarrow$ suffix array + LCP table

1. Suffix tree
2. Suffix array
3. Some applications
4. Finding motifs
Substring motifs of string $T$

- string $T = t_1 \ldots t_n$ in alphabet $A$.
- Problem: what are the frequently occurring (ungapped) substrings of $T$? Longest substring that occurs at least $q$ times?
- Thm: Suffix tree $Tree(T)$ gives complete occurrence counts of all substring motifs of $T$ in $O(n)$ time (although $T$ may have $O(n^2)$ substrings!)

Counting the substring motifs

- internal nodes of $Tree(T)$ $\leftrightarrow$ repeating substrings of $T$
- number of leaves of the subtree of a node for string $P = \text{number of occurrences of } P \text{ in } T$

Substring motifs of hattivatti

Counts for the $O(n)$ maximal motifs shown

Finding repeats in DNA

- human chromosome 3
- the first 48 999 930 bases
- 31 min cpu time (8 processors, 4 GB)
- Human genome: $3 \times 10^9$ bases
- $Tree(\text{HumanGenome})$ feasible

Longest repeat?

<table>
<thead>
<tr>
<th>Occurrences at:</th>
<th>28395800, 26401554r</th>
<th>Length: 2559</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Ten occurrences?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locations: 43525601, 54986723, 14256789, 23456789, 34567890</td>
</tr>
<tr>
<td>Length: 277</td>
</tr>
<tr>
<td>Occurrences at: 10130003, 11421803, 18695837, 26652515, 42971130, 47398125</td>
</tr>
<tr>
<td>In the reversed complement at: 17858493, 41463059, 42431718, 42580925</td>
</tr>
</tbody>
</table>
Using suffix trees: plagiarism

• find longest common substring of strings X and Y
• build Tree(X$Y) and find the deepest node which has a leaf pointing to X and another pointing to Y

Using suffix trees: approximate matching

• edit distance: insertions, deletions, changes
• STOCKHOLM vs TUKHOLMA

String distance/similarity functions

STOCKHOLM vs TUKHOLMA

STOCKHOLM

_ _ _ _ _ _

TUKHOLMA

=> 2 deletions, 1 insertion, 1 change

Dynamic programming

\[
d_{ij} = \min \left( \begin{array}{c}
  \text{if } a_i = b_j \text{ then } d_{i-1,j-1} \\
  d_{i,j-1} + 1 \\
  d_{i-1,j} + 1
\end{array} \right)
\]

= distance between i-prefix of A and j-prefix of B
(substitution excluded)

= optimal alignment by trace-back

Search problem

• find approximate occurrences of pattern P in text T: substrings P' of T such that \( d(P,P') \) small
• dyn progr with small modification: \( O(mn) \)
• lots of (practical) improvement tricks
Index for approximate searching?

- dynamic programming: $P \times \text{Tree}(T)$ with backtracking

Burrows-Wheeler Transformation

- BWT for text compression and indexing

Input: SIX.MIXED.PIXIES.SIFT.SIXTY.PIXIE.DUST.BOXES
Output: TEXYDST.E.IXXIXXSSMP.S..E.X.S.EUSFEDIOIIIT

Burrows-Wheeler

- The method described in the original paper is really a composite of three different algorithms:
  - the block sorting main engine (a lossless, very slightly expansive preprocessor)
  - the move-to-front coder (a byte for byte simple, fast, locally adaptive non-compressor coder)
  - a simple statistical compressor (first order Huffman is mentioned as a candidate) eventually doing the compression.
- Of these three methods only the first two are discussed here as they are what constitutes the heart of the algorithm. These two algorithms combined form a completely reversible (lossless) transformation that - with typical input - skew the first order symbol distributions to make the data more compressible with simple methods. Intuitively speaking, the method transforms slack in the higher order probabilities of the input block (thus making them more even, whitening them) to slack in the lower order statistics. This effect is what is seen in the histogram of the resulting symbol data.
- Please, read the article by Mark Nelson:
  - Data Compression with the Burrows-Wheeler Transform Mark Nelson, Dr. Dobb's Journal
CODE:
t: hat acts like this:<13><10><1

t: hat buffer to the constructor

t: hat corrupted the heap, or we

W: hat goes up must come down<13

t: hat happens, it isn’t likely

w: hat if you want to dynamically

t: hat indicates an error.<13><1

t: hat removes arguments from

t: hat looks like this:<13><10><1

t: hat looks something like this

t: hat looks something like this

t: hat once I detect the mangled

Example

• Decode: errktreteoe.e

• Hint: . Is the last character, alphabetically first...

Burrows-Wheeler Transform

```plaintext
Example
• Decode: errktreteoe.e
• Hint: . Is the last character, alphabetically first...
```