Key take home messages

- Suffix trie, Suffix tree
  - Application examples

- Suffix array

- Burrows Wheeler Transform (BWT)
  - Compact Suffix tree representation: BWT + succinct

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,...,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

O( |P| \log n )
Trie for $D=\{\text{he, hers, his, she}\}$

$S \neq$ 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 $S$.

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

- \( T = \text{hattivatti} \) \( \) ‘text’
- \( P = \text{att} \) \( \) ‘pattern’

- Find the occurrences of \( P \) in \( T \):
  \( \text{hattivatti} \)

- Pattern synthesis:
  \#(t) = 4  \#(atti) = 2
  \#(t****t) = 2


Solution: backtracking with suffix tree

Pattern finding & synthesis problems

- \( T = t_1, t_2, ..., t_n \), \( P = p_1, p_2, ..., p_m \), 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, ...

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: $|\text{Tree}(T)| = O(|T|)$
- 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
  - $|\text{Trie}(T)| = O(|T|^2)$
  - 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
- $v = \text{node}(\alpha)$ if the path from root to $v$ spells $\alpha$
- one-to-one correspondence of leaves and suffixes
- $|T|$ leaves, hence $< |T|$ internal nodes
- $|\text{Tree}(T)| = O(|T| + \text{size}(\text{edge labels}))$
Tree(hattivatti)

substring labels of edges represented as pairs of pointers

Tree(hattivatti)

Tree(T) is full text index

Find att from Tree(hattivatti)

Linear time construction of Tree(T)

On-line construction of Trie(T)

\[ T = t_1t_2 ... t_n \$

\[ P_i = t_1t_2 ... t_i \] \text{ith prefix of } T

\text{on-line idea: update } Trie(P_i) \text{ to } Trie(P_{i+1})

\[ \Rightarrow \text{very simple construction} \]
Trie(abaab)

chain of links connects the end points of current suffixes

Trie(abaab)

Add next symbol = b

From here on b-arc already exists

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

Before

From here on the a_i-arc exists already => stop updating here

After

New nodes
New suffix links

Trie(abaa)

Trie(abaab)

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

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

On-line procedure for suffix trie

1. Create Trie($T_i$): nodes root and $v$, an arc son(root, $t_1 = \alpha$, and suffix links $\text{slink}($v$) := \text{root}$ and $\text{slink}(\text{root}) := \text{root}$
2. for $i := 2$ to $n$ do begin
3. $v_{i-1} := \text{leaf of Trie}(t_i, t_{i-1}$ for string $t_i ... t_1$ (i.e., the deepest leaf)
4. $v := v_{i-1}$; $v' := 0$
5. while node $v$ has no outgoing arc for $t_i$ do begin
6. Create a new node $v''$ and an arc $\text{son}(v, t_i) := v''$
7. if $v'' \neq 0$ then $\text{slink}(v) := v''$
8. $v := \text{slink}(v)$; $v' := v''$
9. for the node $v''$ such that $v'' := \text{son}(v, t_i)$ do
   if $v'' = v'$ then $\text{slink}(v) := \text{root}$ else $\text{slink}(v') := v''$
end

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 Tree($T_i$) 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($T_i$). Then implicit node $(v, \alpha)$ represents node($\alpha'$) of Trie($T_i$)

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 $T$
On-line procedure for suffix tree

Input: string $T = t_1, t_2, \ldots, t_n$
Output: $Tree(T)$

Notation: $son(v, a) = w$ iff there is an arc from $v$ to $w$ with label $a$
$son(v, \epsilon) = v$

Function $Canonize(v, a)$:

```
while $son(v, a') \neq 0$ where $a = a' a''$, $|a'| > 0$
    do
        $v := son(v, a')$;
        $a := a''$
    return $(v, a)$
```

Suffix-tree on-line: main procedure

Create $Tree(t_1)$; $slink(root) := root$
$(v, a) := (root, \epsilon)$ /* $(v, a)$ is the start node $\gamma$*/
for $i := 2$ to $n+1$ do
    $v' := 0$
    while there is no arc from $v$ with label prefix $a'$ do
        if $a' \neq \epsilon$ then 
            divide the arc $w = son(v, a')$ into two $\gamma'$
            $son(v, a) := v''$; $son(v', t_i) := v'''$; $son(v'', \epsilon) := w$
        else
            $son(v, t_i) := v''''$; $v'''' := v$
        if $v'''' \neq 0$ then $slink(v'') := v''''$
        if $v'''' = 0$ then $slink(v'') := v$
        $(v, a) := Canonize(v, a)$
        $(v, a) := Canonize(v, a t_i)$ /* $(v, a)$ = start node of the next round $\gamma$*/

The actual time and space

- $|Tree(T)|$ 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

A general idea can be used to many other forms of approximate search, like Smith-Waterman, position-restricted scoring matrices, regular expression search, etc.

Applications of Suffix Trees

1. **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 $S_i$

Simple analysis task: LCSS

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

Suffix link

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.

Descending suffix walk

Applications of Suffix Trees

1. **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

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

Suffix link

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.

Descending suffix walk

Applications of Suffix Trees

1. **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.
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

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.

Properties of suffix tree... in practice

- Huge overhead due to pointer structure:
  - Standard implementation of suffix tree for human genome requires over 200 GB memory!
  - A careful implementation (using \( \log n \)-bit fields for each value and array layout for the tree) still requires over 40 GB.
  - 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

---

**Suffixes - sorted**

- Sort all suffixes. Allows to perform binary search!

---

**Suffix array: example**

<table>
<thead>
<tr>
<th>1</th>
<th>hattivatti</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>attivatti</td>
</tr>
<tr>
<td>3</td>
<td>tlvivatti</td>
</tr>
<tr>
<td>4</td>
<td>tvivatti</td>
</tr>
<tr>
<td>5</td>
<td>ivatti</td>
</tr>
<tr>
<td>6</td>
<td>vattti</td>
</tr>
<tr>
<td>7</td>
<td>atti</td>
</tr>
<tr>
<td>8</td>
<td>tti</td>
</tr>
<tr>
<td>9</td>
<td>i</td>
</tr>
<tr>
<td>10</td>
<td>ti</td>
</tr>
<tr>
<td>11</td>
<td>ε</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

* suffix array = lexicographic order of the suffixes

---

**Suffix array construction: sort!**

- suffix array = lexicographic order of the suffixes

---

**Suffix array**

- **suffix array** $SA(T)$ = an array giving the lexicographic order of the suffixes of $T$
- space requirement: $5|T|$
- practitioners like suffix arrays (simplicity, space efficiency)
- theoreticians like suffix trees (explicit structure)

---

**Reducing space: suffix array**

- $SA(T)$ = $\{\text{vocab} \mid T \text{ is a word}\}$
- space saving: $O(|T|)$
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>tlivatti</td>
<td>2</td>
</tr>
<tr>
<td>tivatti</td>
<td>1</td>
</tr>
<tr>
<td>ivatti</td>
<td>10</td>
</tr>
<tr>
<td>vatti</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>e</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.

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)

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.

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 \subset [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

0 1 2 3 4 5 6 7 8 9 10 11
- T[0,n) = y a b a d a b b a d o 0 0 ...
- SA = (12,1,6,4,9,3,8,2,7,5,10,11,0)

Step 0: Construct a sample

- for k = 0,1,2
  Bk = {i ∈ [0,n] | i mod 3 = k}
- C = B1 U B2 sample positions
- Sc sample suffixes
- Example: B1 = (1,4,7,10), B2 = (2,5,8,11), C = (1,4,7,10,2,5,8,11)

Step 1: Sort sample suffixes

- for k = 1,2, construct
  Rk = [t1...t4...t8...t10...]
  R = R1 ^ R2 concatenation of R1 and R2
Suffixes of R correspond to Sc: suffix [t1,...,t4,...] corresponds to S1; correspondence is order preserving.
Sort the suffixes of R: radix sort the characters and rename with ranks to obtain R’.
- Example: R = [abb][ada][bba][do0][bba][dab][bad][o00]
  R’ = (1,2,4,6,4,5,3,7)
  SA_R’ = (8,0,1,6,4,2,5,3,7)
  rank(Si) - 1
  4 - 2
  6 - 5
  3 - 7
  8 - 0

Step 1 (cont.)

- once the sample suffixes are sorted, assign a rank to each:
  rank(Si) = the rank of Si in Sc;
  rank(Sn+1) = rank(Sn+2) = 0
- Example:
  R = [abb][ada][bba][do0][bba][dab][bad][o00]
  R’ = (1,2,4,6,4,5,3,7)
  SA_R’ = (8,0,1,6,4,2,5,3,7)
  rank(Si) - 1
  4 - 2
  6 - 5
  3 - 7
  8 - 0

Step 2: Sort nonsample suffixes

- for each non-sample Si ∈ S_{50} (note that rank(Si+1) is always defined for i ∈ B0):
  S_j ≤ S_i ⇔ (t_j,rank(S_j)) ≤ (t_i,rank(S_i+1))
- radix sort the pairs (t_i,rank(S_i+1)).
- Example: S_12 < S_6 < S_9 < 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 ∈ S_{50} with S_j ∈ S_{50}, distinguish two cases:
    - i ∈ B1: S_i ≤ S_j ⇔ (t_i,rank(S_i+1)) ≤ (t_j,rank(S_j+1))
    - i ∈ B2: S_i ≤ S_j ⇔ (t_i,rank(S_i+1)) ≤ (t_j,rank(S_j+1))
  - note that the ranks are defined in all cases!
- S_i ≤ S_j as (a,4) < (a,5) and S_i ≤ S_j 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
  http://www.oed.com/cgi/display/wotd
- PAT index - by Gaston Gonnet (ta on samsi Maple tarkvara üks loojatest ning hiljem molekuulaarbioloogia tarkvarapaketi väljatöötajaid)
- 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!)

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(HumanGenome) feasible

Longest repeat?

Occurrences at: 28395980, 28401554
Length: 2559

Ten occurrences?

Occurrences at: 10130003, 11421803, 18695837, 26652515, 42971130, 47398125
In the reversed complement at: 17858493, 41463059, 42431718, 42580925

Length: 277
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\_\_TU\_KHOLMA

=> 2 deletions, 1 insertion, 1 change

Dynamic programming

\[ d_{i,j} = \min \begin{cases} 
   d_{i-1,j-1} & \text{if } a_i = b_j \\text{ then } d_{i-1,j-1} \text{ else } \infty, \\
   d_{i-1,j} + 1, & d_{i,j-1} + 1 \end{cases} \]

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

<table>
<thead>
<tr>
<th>A</th>
<th>s</th>
<th>t</th>
<th>o</th>
<th>c</th>
<th>k</th>
<th>h</th>
<th>o</th>
<th>l</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>u</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>k</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>h</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>o</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>l</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>m</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>a</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

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 x Tree(T) with backtracking

Burrows-Wheeler Transformation

- BWT for text compression and indexing

Input: SIX_MIXED_PIXIES_SIFT_SIXTY_PIXIE_DUST_BOXES
Output: TEXTDIST_E.IXIXIXXXSMPBS.B.E.S.EUSFDIIIOIIT

Burrows-Wheeler

- The method described in the original paper is really a composite of three different algorithms:
  - the block sorting main engine (a lossy, very slightly expansive preprocessor)
  - the move-to-front coder (a byte-for-byte simple, fast, locally adaptive source/lossy coder)
  - a single 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. Roughly 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:
Example

- **Decode:** errkretreoe.e

- **Hint:** Is the last character, alphabetically first...

CODE:

hat acts like this:<13><10><1
hat buffer to the constructor
hat corrupted the heap, so we
hat goes up must come down:<13
hat happens, it isn't likely
hat if you want to dynamical
hat indicates an error:<13><1
hat it removes arguments from
hat looks like this:<13><155
hat looks something like this
hat looks something like this
hat once I detect the mangled