Indexing techniques

Yuzhen Ye (yye@indiana.edu)
School of Informatics & Computing, IUB
Contents

- We have seen indexing technique—used in BLAST
- Applications that rely on an efficient indexing technique
  - Comparing short reads against reference genomes (e.g., RNA-Seq data analysis)
  - Comparing large genomes (e.g., MUMer)
- Indexing techniques
  - Hash table
  - Suffix tree & suffix array
  - BWT
## Short-read mapping software

<table>
<thead>
<tr>
<th>Software</th>
<th>Technique</th>
<th>Developer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eland</td>
<td>Hashing reads</td>
<td>Illumnia</td>
</tr>
<tr>
<td>SOAP</td>
<td>Hashing refs</td>
<td>BGI</td>
</tr>
<tr>
<td>Maq</td>
<td>Hashing reads</td>
<td>Sanger (Li, Heng)</td>
</tr>
<tr>
<td>Bowtie &amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bowtie2</td>
<td>BWT</td>
<td>Salzberg/UMD</td>
</tr>
<tr>
<td>BWA</td>
<td>BWT</td>
<td>Sanger (Li, Heng)</td>
</tr>
<tr>
<td>SOAP2</td>
<td>BWT &amp; hashing</td>
<td>BGI</td>
</tr>
</tbody>
</table>

All need to use an efficient indexing technique

http://www.oxfordjournals.org/our_journals/bioinformatics/nextgenerationsequencing.html (under Alignment category)
Indexing a text (a genome, etc)

- Example 1: we want to index a genome such that we can look up any k-mer along the genome in $O(1)$ time (without scanning the whole genome).
- Example 2: we want to index a protein database such that we can look up all the proteins containing a word (k-mer) in constant time.
Hashing

- Hashing is an indexing technique that enable fast search by **computing index directly** based on the key.

<table>
<thead>
<tr>
<th>index</th>
<th>key</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>ASTTSA</td>
<td>protein1</td>
</tr>
<tr>
<td>1002</td>
<td>ASTTSS</td>
<td>protein19, protein90</td>
</tr>
</tbody>
</table>

Hash function

<table>
<thead>
<tr>
<th>key</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Some terminologies

- The process of finding a record using some computation to map its key value to a position in the array is called **hashing**.
- The function that maps key values to positions is called a **hash function** \((h)\).
- The array that holds the hash table is called the **hash table** \((HT)\).
- A position in the hash table is also known as a **slot**.
Hash functions and collisions

- The function that maps key values to positions is called a **hash function** \((h)\).
- Typically there are many more values in the key range than there are slots in the hash table.
- Given a hash function \(h\) and two keys \(k_1\) and \(k_2\), if \(h(k_1)=h(k_2)=\beta\), we say that \(k_1\) and \(k_2\) have a **collision** at slot \(\beta\) under hash function \(h\).
- **Perfect hashing** is a system in which records are hashed such that there are no collisions (e.g., indexing k-mers when \(k\) is small).
- An ideal hash function stores the actual records in the collection such that each slot in the hash table has equal probability of being filled; but **clustering** of records happens (many records hash to only a few of the slots)
A simple hash function for integers

- A function used to hash integers to a table of 16 slots

```c
int h(int x) {
    return (x % 16)
}
```

The value returned by this hash function depends solely on the least significant four bits of the key. These bits are likely to be poorly distributed (as an example, a high percentage of the keys might be even numbers, so the low order bit is zero), so the result may also be poorly distributed.
A simple hash function for k-mers

The strings are DNA sequences.

```java
int h(String x, int k):
    b2int = {'A':0, 'T':1, 'C':2, 'G':3}
    add = 1
    idx = 0
    for i in k to 1:
        b = x[i]
        idx = idx + base2int[b] x add
        add = add x 4
    return idx
```

How many slots in the table for k = 3?

k = 40?

h('CA', 2) = 8
Collision resolution

- While the goal of a hash function is to minimize collisions, some collisions are unavoidable in practice.
- Hashing implementations must include some form of collision resolution policy.
- Two class of collision resolution techniques:
  - Open hashing (separate chaining)—collisions are stored outside the table
  - Closed hashing—collisions result in storing one of the records at another slot in the table.
Open hashing

- The simplest form of open hashing defines each slot in the hash table to be the head of a linked list. All records that hash to a particular slot are placed on that slot’s linked list.
- Records within a slot’s list can be ordered in several ways: by insertion order, by key value order, or by frequency-of-access order.
- The average cost for hashing should be $\Theta(1)$; however, if clustering of records exists, then the cost to access a record can be much higher because many elements on the linked list must be searched.
Closed hashing

- Closed hashing stores all records directly in the hash table.
- A collision resolution policy must be built to determine which slot to use when collision is detected.
- The same policy must be followed during search as during insertion.
- Some common closed hashing
  - Bucket hashing --- overflow goes to an overflow bucket
In CS, a suffix tree is a compressed trie containing all the suffixes of the given text as their keys and positions in the text as their values.

Suffix tree allows one to find, extremely efficiently, all distinct subsequences in a given sequence.

There are efficient algorithms to construct suffix trees given by Weiner (1973) and McCreight (1976) (in linear time)

For the task of comparing two DNA sequences, suffix trees allow one to quickly find all subsequences shared by the two inputs.

The genome alignment is then built upon this information.
Suffix tree of a short sequence

An internal node is a repeated sequence in the original string

Leaf is a unique suffix
Matching two sequences

ATCG TA#
  # 7
  A# 6
  TA# 5
  GTA# 4
  CGTA# 3
  TCGTA# 2
  ATCGTA# 1

ATCG AT$
 $ 14
 T$ 13
 AT$ 12
 GAT$ 11
 CGAT$ 10
 TCGAT$ 9
 ATCGAT$ 8

ATCG is the longest common substring

Every unique matching sequence is represented by an internal node with exactly two child nodes, such that the child nodes are leaf nodes from different genomes

Applied in MUMer
MUMer method

- MUMer combines suffix trees, the longest increasing subsequence (LIS) and SW alignment
- Maximal **Unique** Match (MUM) Identification - Identify the longest strings in Genome 1 that have one identical match in Genome 2
  - Naïve method: $O(N^2)$
  - **Using suffix tree**: $O(N)$
- Ordered MUM Selection - Identify the longest set of MUMs such that they occur in order in each of the genomes (using a variation of the well-known algorithm to find the LIS of a sequence of integers)
- Processing Non-matched Regions - Classify non-matched regions as either insertions, SNPs or highly polymorphic regions
A toy example of string (pattern) matching

- **$T = xabxac$**
  - suffixes = \{xabxac, abxac, bxac, xac, ac, c\}
- **Pattern $P_1$: xa**
- **Pattern $P_2$: xb**

Preprocess text $T$, not pattern $P$
Suffix tree for string matching

- Preprocess text T, not pattern P
  - $O(m)$ preprocess time ($m$: the length of the text)
  - $O(n+k)$ search time ($n$: the length of the pattern)
    - $k$ is number of occurrences of P in T

- Match pattern P against tree starting at root until
  - Case 1, P is completely matched
    - Every leaf below this match point is the starting location of P in T
  - Case 2: No match is possible
    - P does not occur in T
Suffix array

Suffix array is a space-efficient data structure, which is more compact than a suffix tree.

The suffix array is basically a sorted array position of all the suffixes of a text. The start positions are sorted in lexicographical (alphabetical) order.

A suffix array for a text of length $n$ can be built in $O(n \log n)$ time,

Searching the text for a pattern of length $m$ can be done in $O(m \log n)$ time by a binary search; reduced to $O(m + \log n)$ if using LCP (longest common prefix).

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td>#</td>
</tr>
</tbody>
</table>

A B R A C A D A B R A #

<table>
<thead>
<tr>
<th>11</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>A#</td>
</tr>
<tr>
<td>7</td>
<td>ABRA#</td>
</tr>
<tr>
<td>0</td>
<td>ABRACADABRA#</td>
</tr>
<tr>
<td>3</td>
<td>ACADABRA#</td>
</tr>
<tr>
<td>5</td>
<td>ADABRA#</td>
</tr>
<tr>
<td>8</td>
<td>BRA#</td>
</tr>
<tr>
<td>1</td>
<td>BRACADABRA#</td>
</tr>
<tr>
<td>4</td>
<td>CADABRA#</td>
</tr>
<tr>
<td>6</td>
<td>DABRA#</td>
</tr>
<tr>
<td>9</td>
<td>RA#</td>
</tr>
<tr>
<td>2</td>
<td>RACADABRA#</td>
</tr>
</tbody>
</table>
Search in suffix array

- Binary search in suffix array: in each of the $O(\log n)$ comparisons, the input pattern $P$ is compared to the current entry of the suffix array, which means a full string comparison of up to $m$ characters (the whole pattern). So the complexity is $O(m\log n)$.

- The complexity is reduced to $O(m + \log n)$ when LCP is used.
**Longest common prefix (LCP)**

$LCP[i]$ is the length of the longest common prefix between the suffixes starting from $SA[i - 1]$ and $SA[i]$. It keeps track of the length of the longest common prefix among two consecutive suffixes of $S$ when arranged in lexicographic order.

How LCP helps?

Assume at one step of the binary search, \([L,...,R]\) is the range of the suffix array with central point \(M\); \(P\) is compared to suffix \(SA[M]\). Assume \(P\) and the corresponding suffix share the first \(k\) characters, and \(P\) is lexicographically larger than suffix \(SA[M]\), so, in the next step, \([M,...,R]\) is considered and a new central point \(M'\) needs to be determined: \(M \ldots \ M' \ldots \ R\).

We know \(lcp(P,M) = k\); also LCP-LR is precomputed such that a O(1)-lookup gives \(lcp(M, M')\), the longest common prefix of \(M\) and \(M'\). Now there are three possibilities:

Case 1: \(k < lcp(M, M')\), i.e. \(P\) has fewer prefix characters in common with \(M\) than \(M\) has in common with \(M'\). This means the \((k+1)\)-th character of \(M'\) is the same as that of \(M\), and since \(P\) is lexicographically larger than \(M\), it must be lexicographically larger than \(M'\), too. So we continue in the right half \([M',...,R]\).

Case 2: \(k > lcp(M, M')\), i.e. \(P\) has more prefix characters in common with \(M\) than \(M\) has in common with \(M'\). Consequently, if we were to compare \(P\) to \(M'\), the common prefix would be smaller than \(k\), and \(M'\) would be lexicographically larger than \(P\), so, without actually making the comparison, we continue in the left half \([M,...,M']\).

Case 3: \(k == lcp(M, M')\). So \(M\) and \(M'\) are both identical with \(P\) in the first \(k\) characters. To decide whether we continue in the left or right half, it suffices to compare \(P\) to \(M'\) starting from the \((k+1)\)-th character.

The consequence is that no character of \(P\) is compared to any character of the text more than once. The total number of character comparisons is bounded by \(m\), so the total complexity is indeed \(O(m+\log n)\).

Burrows-Wheeler Transform

- Burrow M & Wheeler D (1994)
- Reversible permutation of text to allow better compression (e.g. bzip2)
- Algorithms exist to perform fast search on BW-transformed data
Burrows-Wheeler Transform (BWT)

Suffix array

sorted in lexicographical order

Burrows-Wheeler Matrix (BWM)
Why Burrows-Wheeler?

- BWT very compact (was developed for compression purpose)
  - Approximately $\frac{1}{2}$ byte per base (by contrast, an integer number may take 4 bytes)
  - As large as the original text, plus a few “extras” (the FM indices)
  - Can fit onto a standard computer with 2GB of memory (for indexing a human genome)

- Linear-time search algorithm
  - Proportional to the length of query for exact matches
Mississippi example

<table>
<thead>
<tr>
<th>m</th>
<th>i</th>
<th>s</th>
<th>s</th>
<th>i</th>
<th>s</th>
<th>s</th>
<th>i</th>
<th>p</th>
<th>p</th>
<th>i</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>$</td>
<td>m</td>
</tr>
<tr>
<td>s</td>
<td>s</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>$</td>
<td>m</td>
</tr>
<tr>
<td>s</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>$</td>
<td>m</td>
<td>i</td>
</tr>
<tr>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>$</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
</tr>
<tr>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>$</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>s</td>
</tr>
<tr>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>$</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>i</td>
</tr>
<tr>
<td>i</td>
<td>p</td>
<td>p</td>
<td>i</td>
<td>$</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>s</td>
</tr>
<tr>
<td>p</td>
<td>p</td>
<td>i</td>
<td>$</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>s</td>
<td>s</td>
</tr>
<tr>
<td>p</td>
<td>i</td>
<td>$</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>p</td>
</tr>
<tr>
<td>i</td>
<td>$</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>m</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>s</td>
<td>s</td>
<td>i</td>
<td>p</td>
<td>p</td>
</tr>
</tbody>
</table>
Sorted lexicographically

\[ \mathcal{F} \]

\[
\begin{array}{cccccccccccc}
\$ & m & i & s & s & i & s & s & i & p & p & i \\
i & \$ & m & i & s & s & i & s & s & i & p & p \\
i & p & p & i & \$ & m & i & s & s & i & s & s \\
i & s & s & i & p & p & i & \$ & m & i & s & s \\
i & s & s & s & i & s & s & i & p & p & i & \$ \\
m & i & s & s & s & i & s & s & i & p & p & i \\
p & i & \$ & m & i & s & s & i & s & s & s & i \\
p & p & p & i & \$ & m & i & s & s & i & s & s \\
s & i & p & p & i & \$ & m & i & s & s & s & i \\
s & i & s & s & i & p & p & i & \$ & m & i & s \\
s & s & s & i & p & p & i & \$ & m & i & s & s & i \\
s & s & i & s & s & i & p & p & i & \$ & m & i \\
\end{array}
\]
### FM indices

**C table:** $C[c]$ # of text characters which are alphabetically smaller than c

$$\begin{array}{cccccc}
\$ & m & i & s & s & i \\
i & $ & m & i & s & s & i & p \\
i & p & p & i & $ & m & i & s & s \\
i & s & s & i & p & p & i & $ & m & s \\
i & s & s & i & s & s & i & p & p & i \\
i & m & s & s & i & s & s & i & p & p & i \\
p & i & $ & m & i & s & s & i & s & i \\
p & p & i & $ & m & i & s & s & s & i \\
s & i & p & p & i & $ & m & i & s & s \\
s & s & s & i & p & p & i & $ & m & s & i \\
s & s & i & s & s & i & p & p & i & $ & m \\
s & s & i & s & s & i & p & p & i & $ & m & s \\
s & s & s & i & p & p & i & $ & m & s & i & s \\
s & s & i & s & s & i & p & p & i & $ & m & s & i \\
s & s & i & s & s & i & p & p & i & $ & m & s & i & s \\
\end{array}$$

**Occ function**

$Occ(c,q)$: # of occurrences of character $c$ in the prefix $T_{bw}[1,q]$

- $Occ('s',10) = 4$ (4x‘s’ in “ipssm$piss$”)

**C[‘m’] = 5 (4x‘i’ + ‘$’)**

| $|$ | i | m | p | s |
|----|---|---|---|---|
| 0  | 1 | 6 | 8 | 6 |

**SA intervals??**

| $|$ | i | m | p | s |
|----|---|---|---|---|
| 0  | 1 | 0 | 0 | 0 |
| 0  | 1 | 0 | 1 | 0 |
| 0  | 1 | 0 | 1 | 1 |
| 0  | 1 | 0 | 1 | 2 |
| 0  | 1 | 1 | 1 | 2 |
| 1  | 1 | 2 | 2 | 3 |
| 1  | 2 | 2 | 4 | 4 |
| 1  | 4 | 1 | 2 | 4 |
**Last to front mapping**

$LF() = \text{last to front mapping}: \text{The character } T_{bw}[i] \text{ is located in the first column } F \text{ at position } LF[i]; \text{ i.e., } T_{bw}[i] = F[LF[i]]$

Suffix-id prior to the first $T_{bw}[i]$:

$$LF(i)=C[T_{bw}[i]] + \text{Occ}(T_{bw}[i], i)$$

e.g.,

$$LF(10)=C['s'] + \text{Occ}(‘s’, 10) = 8 + 4 = 12$$

Both $T_{bw}[10]$ and $F[12]$ correspond to the first ‘s’ in “mississippi”

<table>
<thead>
<tr>
<th>$F$</th>
<th>$T_{bw}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>i</td>
<td>mississippi</td>
</tr>
<tr>
<td>p</td>
<td>mississippi</td>
</tr>
<tr>
<td>s</td>
<td>mississippi</td>
</tr>
<tr>
<td>i</td>
<td>mississippi</td>
</tr>
<tr>
<td>p</td>
<td>mississippi</td>
</tr>
<tr>
<td>i</td>
<td>mississippi</td>
</tr>
<tr>
<td>s</td>
<td>mississippi</td>
</tr>
<tr>
<td>s</td>
<td>mississippi</td>
</tr>
<tr>
<td>s</td>
<td>mississippi</td>
</tr>
<tr>
<td>s</td>
<td>mississippi</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
</tr>
</tbody>
</table>

Only $T_{bw}$ is stored; $F$ is shown for demonstration purpose
Reversible transform (using LF mapping) to recover text

Text: acaacg$

C table
<table>
<thead>
<tr>
<th>Text</th>
<th>g</th>
<th>c</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>c</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>g</td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>c</td>
<td>g</td>
<td>$</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>g</td>
<td>$</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>g</td>
<td>$</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>g</td>
<td>$</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>g</td>
<td>$</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>c</td>
<td>g</td>
<td>$</td>
<td>a</td>
</tr>
</tbody>
</table>

Occ table
<table>
<thead>
<tr>
<th>#</th>
<th>$</th>
<th>a</th>
<th>c</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

LF(0) = C[‘g’] + Occ(‘g’, 0) = 6 + 1 = 7

LF(6) = C[‘c’] + Occ(‘c’, 6) = 4 + 2 = 6
Searching in BWT-compressed text: counting the occurrences

Algorithm for counting the number of occurrences of $P[1; p]$ in $T[1; u]$; **backward search** algorithm

Algorithm `BW_Search(P[1, p])`

1. $c = P[p], i = p$;
2. $sp = C[c] + 1, ep = C[c + 1]$;
3. while ($sp \leq ep$ and $(i \geq 2)$) do
   4. $c = P[i - 1]$;
   5. $sp = C[c] + \text{Occ}(c, 1, sp - 1) + 1$;
   6. $ep = C[c] + \text{Occ}(c, 1, ep)$;
   7. $i = i - 1$;
4. if ($ep < sp$) then return “not found” else return “found ($ep - sp + 1$) occurrences”.

Ferragina and Manzini showed that it is possible to compute $\text{Occ}(c, 1, k)$ in constant time. $\text{Occ}(c, 1, k)$ sometimes is shown as $\text{Occ}(c, k) : \#$ of occurrences of character $c$ in the prefix $1$ to $k$. At the $i$-th phase, the parameter $sp$ points to the first row of $M$ prefixed by $P[i; p]$ and the parameter $ep$ points to the last row of $M$ prefixed by $P[i; p]$. 
Searching in BWT-compressed text: determining the occurrences (locations)

- We know how to count the occurrences, then for \( s = sp, \ sp + 1, \ldots, \ ep \), we need to find the position \( \text{pos}(s) \) in \( T \) of the suffix which prefixes the \( sth \) row \( M[s] \); two methods were proposed.

- The first one is simple, which relies on a subset of the indices in \( T^{bw} \) that are associated with a position in suffix array. If \( T^{bw}[j] \) has a position associated with it, \( \text{locate}(j) \) is trivial. If it's not associated, the string is followed with \( \text{LF}(i) \) (last front mapping) until an associated index is found. \( \text{Locate} \) can be implemented to find \( \text{occ} \) occurrences of a pattern \( P[1..p] \) in a text \( T[1..u] \) in \( O(p + \text{occ} \log^\epsilon u) \) time.

- The second method is faster and relies on the very special properties of the the string \( T^{bw} \) and on a different compression algorithm.

---

Ferragina and Manzini, 2000 “Opportunistic Data Structures with Applications”.
Exact match: A simple example

Exact match (another example)

BWT(agcagcagact) = tgcc$ggaaaac

Search for pattern: gca

gca

$agcagcagact
act$agcagcag
agact$agcagc
agcagact$agc
agcagcagact$

$agcagcagact
act$agcagcag
agact$agcagc
agcagact$agc
agcagcagact$

$agcagcagact
act$agcagcag
agact$agcagc
agcagact$agc
agcagcagact$

$agcagcagact
act$agcagcag
agact$agcagc
agcagact$agc
agcagcagact$

Test with your own seq and pattern at: http://www.allisons.org/ll/AlgDS/Strings/BWT/
Inexact match

- For the exact matching problem, we only need to check one SA interval; for the inexact matching problem, there may be many.
Main advantage of BWT against suffix array

- BWT needs less memory than suffix array
- For human genome $m = 3 \times 10^9$ ($m$ is the length of the genome):
  - Suffix array: $m \log_2(m)$ bits $= 4m$ bytes $= 12$GB
  - BWT: $m/4$ bytes plus extras $= 1 - 2$ GB
    - $m/4$ bytes to store BWT ($2$ bits per char)
    - Suffix array and occurrence counts array take a lot more bits
    - In practice, SA and OCC only partially stored, most elements are computed on demand (takes time!)
    - Tradeoff between time and space
# Short-read mapping software

<table>
<thead>
<tr>
<th>Software</th>
<th>Technique</th>
<th>Developer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eland</td>
<td>Hashing reads</td>
<td>Illumnia</td>
</tr>
<tr>
<td>SOAP</td>
<td>Hashing refs</td>
<td>BGI</td>
</tr>
<tr>
<td>Maq</td>
<td>Hashing reads</td>
<td>Sanger (Li, Heng)</td>
</tr>
<tr>
<td>Bowtie &amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bowtie2</td>
<td>BWT</td>
<td>Salzberg/UMD</td>
</tr>
<tr>
<td>BWA</td>
<td>BWT</td>
<td>Sanger (Li, Heng)</td>
</tr>
<tr>
<td>SOAP2</td>
<td>BWT &amp; hashing</td>
<td>BGI</td>
</tr>
</tbody>
</table>