Succinct Data Structures for NLP-at-Scale

Matthias Petri       Trevor Cohn

Computing and Information Systems
The University of Melbourne, Australia
first.last@unimelb.edu.au

November 20, 2016
<table>
<thead>
<tr>
<th>Trevor Cohn, University of Melbourne</th>
</tr>
</thead>
<tbody>
<tr>
<td>■ Probabilistic machine learning for structured problems in language: NP Bayes, Deep learning, etc.</td>
</tr>
<tr>
<td>■ Applications to machine translation, social media, parsing, summarisation, multilingual transfer.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Matthias Petri, University of Melbourne</th>
</tr>
</thead>
<tbody>
<tr>
<td>■ Data Compression, Succinct Data Structures, Text Indexing, Compressed Text Indexes, Algorithmic Engineering, Terabyte scale text processing</td>
</tr>
<tr>
<td>■ Machine Translation, Information Retrieval, Bioinformatics</td>
</tr>
</tbody>
</table>
Who are we?

Tutorial based partly on research [Shareghi et al., 2015, Shareghi et al., 2016b] with collaborators at Monash University:

Ehsan Shareghi  
Gholamreza Haffari
1. Introduction and Motivation (15 Minutes)

2. Basic Technologies and Notation (20 Minutes)

3. Index based Pattern Matching (20 Minutes)

4. Pattern Matching using Compressed Indexes (40 Minutes)

5. Applications to NLP (30 Minutes)
Introduction and Motivation (15 Mins)

1. What
2. Why
3. Who and Where
What is it?

- Data structures and algorithms for working with large data sets
- Desiderata
  - minimise space requirement
  - maintaining efficient searchability
- Classes of compression do just this! Near-optimal compression, with minor effect on runtime
- E.g., bitvector and integer compression, wavelet trees, compressed suffix array, compressed suffix trees
Why do we need it?

- Era of ‘big data’: text corpora are often 100s of gigabytes or terabytes in size (e.g., CommonCrawl, Twitter)
- Even simple algorithms like counting $n$-grams become difficult
- One solution is to use distributed computing, however can be very inefficient
- Succinct data structures provide a compelling alternative, providing compression and efficient access
- Complex algorithms become possible in memory, rather than requiring cluster and disk access
Why do we need it?

- Era of ‘big data’: text corpora are often 100s of gigabytes or terabytes in size (e.g., CommonCrawl, Twitter)
- Even simple algorithms like counting $n$-grams become difficult
- One solution is to use distributed computing, however can be very inefficient
- Succinct data structures provide a compelling alternative, providing compression and efficient access
- Complex algorithms become possible in memory, rather than requiring cluster and disk access

E.g., Infinite order language model possible, with runtime similar to current fixed order models, and lower space requirement.
Who uses it and where is it used?

Surprisingly few applications in NLP
- Bioinformatics, Genome assembly
- Information Retrieval, Graph Search (Facebook)
- Search Engine Auto-complete
- Trajectory compression and retrieval
- XML storage and retrieval (xpath queries)
- Geo-spatial databases
- ...

Basic Technologies and Notation (20 Mins)

1. Bitvectors
2. Rank and Select
3. Succinct Tree Representations
4. Variable Size Integers
Basic Building blocks: the bitvector

**Definition**

A bitvector (or bit array) $B$ of length $n$ compactly stores $n$ binary numbers using $n$ bits.

**Example**

<table>
<thead>
<tr>
<th></th>
<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>$B$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

# Bitvector operations

## Access and Set

\[ B[0] = 1, \ B[0] = B[1] \]

## Logical Operations

\[ A \ OR \ B, \ A \ AND \ B, \ A \ XOR \ B \]

## Advanced Operations

- POPCOUNT\( (B) \): Number of one bits set
- MSB\_SET\( (B) \): Most significant bit set
- LSB\_SET\( (B) \): Least significant bit set
Operation **Rank**

**Definitions**

\[ \text{Rank}_1(B, j) : \text{How many 1's are in } B[0, j] \]

\[ \text{Rank}_0(B, j) : \text{How many 0's are in } B[0, j] \]

**Example**

\[
B = \begin{bmatrix}
1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0
\end{bmatrix}
\]

\[ \text{Rank}_1(B, 7) = 5 \]
\[ \text{Rank}_0(B, 7) = 8 - \text{Rank}_1(B, 7) = 3 \]
Operation **SELECT**

**Definitions**

\[ \text{SELECT}_1(B, j): \text{ Where is the } j\text{-th (start count at 0) 1 in } B \]

\[ \text{SELECT}_0(B, j): \text{ Where is the } j\text{-th (start count at 0) 0 in } B \]

Inverse of **RANK**: \[ \text{RANK}_1(B, \text{SELECT}_1(B, j)) = j \]

**Example**

\[ B = \begin{array}{cccccccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
\hline
1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
\end{array} \]

\[ \text{SELECT}_1(B, 4) = 7 \]

\[ \text{SELECT}_0(B, 3) = 8 \]
# Complexity of Operations \textbf{Rank} and \textbf{Select}

## Simple and Slow

Scan the whole bitvector using $O(1)$ extra space and $O(n)$ time to answer both \textbf{Rank} and \textbf{Select}.

## Constant time \textbf{Rank}

Periodically store the absolute count up till that position explicitly. Only scan a small part of the bitvector to get the right answer. Space usage: $n + o(n)$ bits. Runtime: $O(1)$. In practice: 25% extra space.

## Constant time \textbf{Select}

Similar to \textbf{Rank} but more complex as blocks are based on the number of 1/0 observed.
Compressed Bitvectors

Idea

If only few 1’s or clustering present in the bitvector, we can use compression techniques to substantially reduce space usage while efficiently supporting operations Rank and Select.

In Practice

Bitvector of size 1 GiB with 10% of all bits randomly set to 1:

- Encodings:
  - Elias-Fano ['73]: $x$ MiB
  - RRR ['02]: $y$ MiB
Bitvectors - Practical Performance

How fast are \texttt{Rank} and \texttt{Select} in practice? Experiment: Cost per operation averaged over 1M executions: \texttt{(code)}

Uncompressed:

<table>
<thead>
<tr>
<th>BV Size</th>
<th>Access</th>
<th>Rank</th>
<th>Select</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>1MB</td>
<td>3ns</td>
<td>4ns</td>
<td>47ns</td>
<td>127%</td>
</tr>
<tr>
<td>10MB</td>
<td>10ns</td>
<td>14ns</td>
<td>85ns</td>
<td>126%</td>
</tr>
<tr>
<td>1GB</td>
<td>26ns</td>
<td>36ns</td>
<td>303ns</td>
<td>126%</td>
</tr>
<tr>
<td>10GB</td>
<td>78ns</td>
<td>98ns</td>
<td>372ns</td>
<td>126%</td>
</tr>
</tbody>
</table>

Compressed:

<table>
<thead>
<tr>
<th>BV Size</th>
<th>Access</th>
<th>Rank</th>
<th>Select</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>1MB</td>
<td>68ns</td>
<td>65ns</td>
<td>49ns</td>
<td>33%</td>
</tr>
<tr>
<td>10MB</td>
<td>99ns</td>
<td>88ns</td>
<td>58ns</td>
<td>30%</td>
</tr>
<tr>
<td>1GB</td>
<td>292ns</td>
<td>275ns</td>
<td>219ns</td>
<td>32%</td>
</tr>
<tr>
<td>10GB</td>
<td>466ns</td>
<td>424ns</td>
<td>336ns</td>
<td>30%</td>
</tr>
</tbody>
</table>
Using **Rank and Select**

- Basic building block of many compressed / succinct data structures
- Different implementations provide a variety of time and space trade-offs
- Implemented an ready to use in SDSL and many others:
  - [http://github.com/simongog/sdsl-lite](http://github.com/simongog/sdsl-lite)
  - [http://github.com/facebook/folly](http://github.com/facebook/folly)
  - [http://sux.di.unimi.it](http://sux.di.unimi.it)
  - [http://github.com/ot/succinct](http://github.com/ot/succinct)
- Used in practice! For example: Facebook Graph search (Unicorn)
Succinct Tree Representations

Idea

Instead of storing pointers and objects, flatten the tree structure into a bitvector and use \texttt{RANK} and \texttt{SELECT} to navigate

From

\begin{verbatim}
typedef struct {
    void* data;  // 64 bits
    node_t* left; // 64 bits
    node_t* right; // 64 bits
    node_t* parent; // 64 bits
} node_t;
\end{verbatim}

To

\texttt{Bitvector + RANK + SELECT + Data (\approx 2 \text{ bits per node})}
**Definition: Succinct Data Structure**

A succinct data structure uses space “close” to the information theoretical lower bound, but still supports operations time-efficiently.

**Succinct Tree Representations:**

There number of unique binary trees containing $n$ nodes is (roughly) $4^n$. To differentiate between them we need at least $\log_2(4^n) = 2n$ bits. Thus, a succinct tree representations should require $2n$ bits (plus a bit more).
LOUDS level order unary degree sequence

A succinct representation of a rooted, ordered tree containing nodes with arbitrary degree [Jacobson’89]

Example:
Add Pseudo Root:
LOUDS Step 2

For each node unary encode the number of children:

Unary encoding of out degree → 01

pseudo root
Write out unary encodings in level order:

LOUDS sequence $L = 0100010011010101111$
Each node (except the pseudo root) is represented twice
- Once as “0” in the child list of its parent
- Once as the terminal (“1”) in its child list

Represent node $v$ by the index of its corresponding “0”
- I.e. root corresponds to “0”

A total of $2n$ bits are used to represent the tree shape!
## LOUDS Navigation

Use **Rank** and **Select** to navigate the tree in constant time

### Examples:

**Compute node degree**

```c++
int node_degree(int v) {
    if is_leaf(v) return 0
    id = RANK₀(L, v)
    return SELECT₁(L, id + 2) - SELECT₁(L, id + 1) - 1
}
```

**Return the \( i \)-th child of node \( v \)**

```c++
int child(int v, i) {
    if i > node_degree(v) return -1
    id = RANK₀(L, v)
    return SELECT₁(L, id + 1) + i
}
```

Complete construction, load, storage and navigation code of LOUDS is only 200 lines of C++ code.
Using 32 or 64 bit integers to store mostly small numbers is wasteful

Many efficient encoding schemes exist to reduce space usage
Variable Byte Compression

Idea

Use variable number of bytes to represent integers. Each byte contains 7 bits “payload” and one continuation bit.

Examples

<table>
<thead>
<tr>
<th>Number</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>824</td>
<td>00000110 10111000</td>
</tr>
<tr>
<td>5</td>
<td>10000101</td>
</tr>
</tbody>
</table>

Storage Cost

<table>
<thead>
<tr>
<th>Number Range</th>
<th>Number of Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 – 127</td>
<td>1</td>
</tr>
<tr>
<td>128 – 16383</td>
<td>2</td>
</tr>
<tr>
<td>16384 – 2097151</td>
<td>3</td>
</tr>
</tbody>
</table>
# Variable Byte Compression - Algorithm

## Encoding

1. **function** `ENCODE(x)`
2. **while** `x >= 128` **do**
3. `WRITE(x mod 128)`
4. `x = x ÷ 128`
5. **end while**
6. `WRITE(x + 128)`
7. **end function**

## Decoding

1. **function** `DECODE(bytes)`
2. `x = 0`
3. `y = READDYTE(bytes)`
4. **while** `y < 128` **do**
5. `x = 128 × x + y`
6. `y = READDYTE(bytes)`
7. **end while**
8. `x = 128 × x + (y - 128)`
9. **return** `x`
10. **end function**
Variable Sized Integer Sequences

Problem

Sequences of vbyte encoded numbers can not be accessed at arbitrary positions

Solution: Directly addressable variable-length codes (DAC)

Separate the indicator bits into a bitvector and use \texttt{Rank} and \texttt{Select} to access integers in $O(1)$ time. [Brisboa et al.’09]
DAC - Concept

Sample vbyte encoded sequence of integers:

```
01010101 1110111 11000111 00110110 01110110 10000100 11101011 10000110 01110110 10000001 10000000 10001000
```

DAC restructuring of the vbyte encoded sequence of integers:

```
01010101 11000111 00110110 11101011 10000110 01110110 10000000 10001000
1110111 01110110 10000001
10000100
```

Separate the indicator bits:

```
01011111 1000111 0110110 1101011 0000110 1101011 0000000 0001000
101
1110111 1110110 0000001
1 0000100
```
DAC - Access

Accessing element A[5]:

- Access indicator bit of the first level at position 5: $I_1[5] = 0$
- 0 in the indicator bit implies the number uses at least 2 bytes
- Perform $Rank_0(I_1, 5) = 3$ to determine the number of integers in $A[0, 5]$ with at least two bytes
- Access payloads and recover number in $O(1)$ time.
<table>
<thead>
<tr>
<th>Bitvectors</th>
<th>Rank and Select</th>
<th>Succinct Tree Representations</th>
<th>Variable Size Integers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Practical Exercise**
Index based Pattern Matching (20 Mins)

5  Suffix Trees

6  Suffix Arrays

7  Compressed Suffix Arrays
Pattern Matching

Definition
Given a text $T$ of size $n$, find all occurrences (or just count) of pattern $P$ of length $m$.

Online Pattern Matching
Preprocess $P$, scan $T$. Examples: KMP, Boyer-Moore, BMH etc. $O(n + m)$ search time.

Offline Pattern Matching
Preprocess $T$, Build Index. Examples: Inverted Index, Suffix Tree, Suffix Array. $O(m)$ search time.
Suffix Tree (Weiner’73)

- Data structure capable of processing $T$ in $O(n)$ time and answering search queries in $O(n)$ space and $O(m)$ time. Optimal from a theoretical perspective.
- All suffixes of $T$ into a trie (a tree with edge labels)
- Contains $n$ leaf nodes corresponding to the $n$ suffixes of $T$
- Search for a pattern $P$ is performed by finding the subtree corresponding to all suffixes prefixed by $P$
Suffix Tree - Example

\[ T = \text{abracadabra} \text{cara} \text{b}\$ \]
**Suffix Tree - Example**

\[ T = \text{abracadabra} \text{bracarab$} \]

**Suffixes:**

<table>
<thead>
<tr>
<th>Index</th>
<th>Suffix</th>
</tr>
</thead>
</table>
| 0     | abracadabraabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbabracbbr


Suffix Tree - Example

The diagram illustrates a suffix tree for the string "rabca$.d..$". Each node represents a substring, and the numbers indicate the suffix array positions for each substring. The tree is structured to efficiently store and search for suffixes of the input string.
Suffix Tree - Search for "aca"
Space usage in practice is large. 20 – 40 times $n$ for highly optimized implementations.

Only usable for small datasets.
Suffix Arrays (Manber’89)

- Reduce space of Suffix Tree by only storing the $n$ leaf pointers into the text
- Requires $n \log n$ bits for the pointers plus $T$ to perform search
- In practice $5 - 9n$ bytes for character alphabets
- Search for $P$ using binary search
Suffix Arrays - Example

\[ T = \text{abracadabra\$} \]
### Suffix Arrays - Example

\[ T = \text{abracadabracarab}\$

**Suffixes:**

<table>
<thead>
<tr>
<th>Suffix Index</th>
<th>Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>abracadabracarab$</td>
</tr>
<tr>
<td>1</td>
<td>bracadabracarab$</td>
</tr>
<tr>
<td>2</td>
<td>racadabracarab$</td>
</tr>
<tr>
<td>3</td>
<td>acadabracarab$</td>
</tr>
<tr>
<td>4</td>
<td>cadabracarab$</td>
</tr>
<tr>
<td>5</td>
<td>adabracarab$</td>
</tr>
<tr>
<td>6</td>
<td>dabracarab$</td>
</tr>
<tr>
<td>7</td>
<td>abracarab$</td>
</tr>
<tr>
<td>8</td>
<td>bracarab$</td>
</tr>
<tr>
<td>9</td>
<td>racarab$</td>
</tr>
<tr>
<td>10</td>
<td>acarab$</td>
</tr>
<tr>
<td>11</td>
<td>carab$</td>
</tr>
<tr>
<td>12</td>
<td>arab$</td>
</tr>
<tr>
<td>13</td>
<td>rab$</td>
</tr>
<tr>
<td>14</td>
<td>ab$</td>
</tr>
<tr>
<td>15</td>
<td>b$</td>
</tr>
<tr>
<td>16</td>
<td>$</td>
</tr>
</tbody>
</table>
Suffix Arrays - Example

\[ T = \text{abracadabracarab}\$ \]

Sorted Suffixes:

<table>
<thead>
<tr>
<th>Index</th>
<th>Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>$</td>
</tr>
<tr>
<td>14</td>
<td>ab$</td>
</tr>
<tr>
<td>0</td>
<td>abracadabracarab$</td>
</tr>
<tr>
<td>7</td>
<td>abracarab$</td>
</tr>
<tr>
<td>3</td>
<td>acadabracarab$</td>
</tr>
<tr>
<td>10</td>
<td>acarab$</td>
</tr>
<tr>
<td>5</td>
<td>adabracarab$</td>
</tr>
<tr>
<td>12</td>
<td>arab$</td>
</tr>
<tr>
<td>15</td>
<td>b$</td>
</tr>
<tr>
<td>1</td>
<td>bracadabracarab$</td>
</tr>
<tr>
<td>8</td>
<td>bracarab$</td>
</tr>
<tr>
<td>4</td>
<td>cadabracarab$</td>
</tr>
<tr>
<td>11</td>
<td>carab$</td>
</tr>
<tr>
<td>6</td>
<td>dabracarab$</td>
</tr>
<tr>
<td>13</td>
<td>rab$</td>
</tr>
<tr>
<td>2</td>
<td>racadabracarab$</td>
</tr>
<tr>
<td>9</td>
<td>racarab$</td>
</tr>
</tbody>
</table>
Suffix Arrays - Example

\[ T = \text{abracadabra} \]$
Suffix Arrays - Search

\[ T = \text{abracadabracarab}$, \ P = \text{abr} \]
Suffix Arrays - Search

\[ T = \text{abracadabra} \text{bracarab}, \quad P = \text{abr} \]
$T = \text{abracadabra\textbackslash \textasciitilde}, \ P = \text{abr}$

<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>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</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>c</td>
<td>a</td>
<td>r</td>
<td>a</td>
<td>b</td>
<td>b</td>
</tr>
</tbody>
</table>

<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>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>14</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>10</td>
<td>5</td>
<td>12</td>
<td>15</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>11</td>
<td>6</td>
<td>13</td>
<td>2</td>
<td>9</td>
</tr>
</tbody>
</table>
$T = \text{abracadabracarab}$, $P = \text{abr}$
Suffix Arrays - Search

\[ T = \text{abracadabracarab}$, \]

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
\text{a} & \text{b} & \text{r} & \text{a} & \text{c} & \text{a} & \text{d} & \text{a} & \text{b} & \text{r} & \text{a} & \text{c} & \text{a} & \text{r} & \text{a} & \text{b} & \text{b} \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
16 & 14 & 0 & 7 & 3 & 10 & 5 & 12 & 15 & 1 & 8 & 4 & 11 & 6 & 13 & 2 & 9 \\
\end{array}
\]
In practice:

- Suffix Trees requires $\approx 20n$ bytes of space (for efficient implementations)
- Suffix Arrays require $5 - 9n$ bytes of space
- Comparable search performance

Example: 5GB English text requires 45GB for a character level suffix array index and up to 200GB for suffix trees
Suffix Arrays / Trees - Construction

In theory: Both can be constructed in optimal $O(n)$ time

In practice:

- Suffix Trees and Suffix Arrays construction can be parallelized
- Most efficient suffix array construction algorithm in practice are note $O(n)$
- Efficient semi-external memory construction algorithms exist
- Parallel suffix array construction algorithms can index 20MiB/s (24 threads) in-memory and 4MiB/s in external memory
- Suffix Arrays of terabyte scale text collection can be constructed. Practical!
- Word-level Suffix Array construction also possible.
Dilemma

- There is lots of work out there which proposes solutions for different problems based on suffix trees.
- Suffix trees (and to a certain extent suffix arrays) are not really applicable for large scale problems.
- However, large scale suffix arrays can be constructed efficiently without requiring large amounts of memory.

Solutions:

- External or Semi-External memory representation of suffix trees / arrays.
There is lots of work out there which proposes solutions for different problems based on suffix trees.

Suffix trees (and to a certain extend suffix arrays) are not really applicable for large scale problems.

However, large scale suffix arrays can be constructed efficiently without requiring large amounts of memory.

Solutions:

- External or Semi-External memory representation of suffix trees / arrays
- Compression?
External / Semi-External Suffix Indexes

String-B Tree

- Cache-Oblivious
- Complicated
- Not implemented anywhere (not practical?)
Compressed Suffix Arrays and Trees

Idea

Utilize data compression techniques to substantially reduce the space of suffix arrays/trees while retaining their functionality.

Compressed Suffix Arrays (CSA):

- Use space equivalent to the compressed size of the input text. Not 4-8 times more! Example: 1GB English text compressed to roughly 300MB using gzip. CSA uses roughly 300MB (sometimes less).
- Provide more functionality than regular suffix arrays.
- Implicitly contain the original text, no need to retain it. Not needed for query processing.
- Similar search efficiency than regular suffix arrays.
- Used to index terabytes of data on a reasonably powerful machine!
#include "sds1/suffix_arrays.hpp"
#include <iostream>

int main(int argc, char ** argv) {
    std::string input_file = argv[1];
    std::string out_file = argv[2];
    sds1::csa WT csa;
    sds1::construct(csa, input_file, 1);
    std::cout << "CSA size = "
              << sds1::size_in_megabytes(csa) << std::endl;
    sds1::store_to_file(csa, out_file);
}

How does it work? Find out after the break!
Break Time

See you back here in 20 minutes!
Compressed Indexes (40 Mins)

1. CSA Internals
2. BWT
3. Wavelet Trees
4. CSA Usage
5. Compressed Suffix Trees
Two practical approaches developed independently:

- **CSA-WT**: Also referred to as the FM-Index. Proposed by Ferragina and Manzini in 2000.

Many practical (and theoretical) improvements to compression, query speed since then. Efficient implementations available in SDSL: `csa_sada<>` and `csa_wt<>`.

For now, we focus on CSA-WT.
CSA-WT or the FM-Index

- Utilizes the Burrows-Wheeler Transform (BWT) used in compression tools such as bzip2

- Requires \texttt{Rank} and \texttt{Select} on non-binary alphabets

- Heavily utilize compressed bitvector representations

- Theoretical bound on space usage related to compressibility (entropy) of the input text
The Burrows-Wheeler Transform (BWT)

- Reversible Text Permutation

- Initially proposed by Burrows and Wheeler as a compression tool. The BWT is more compressible than the original text!

- Defined as $BWT[i] = T[SA[i] - 1 \mod n]$

- In words: $BWT[i]$ is the symbol preceding suffix $SA[i]$ in $T$

Why does it work? How is it related to searching?
BWT - Example

\[ T = \text{abracadabracarab} \]$
BWT - Example

\[ T = \text{abracadabracarab}\$

\begin{align*}
0 & : \text{abracadabracarab}\$
1 & : \text{bracadabracarab}\$
2 & : \text{racadabracarab}\$
3 & : \text{acadabracarab}\$
4 & : \text{cadabracarab}\$
5 & : \text{adabracarab}\$
6 & : \text{dabracarab}\$
7 & : \text{abracarab}\$
8 & : \text{bracarab}\$
9 & : \text{racarab}\$
10 & : \text{acarab}\$
11 & : \text{carab}\$
12 & : \text{arab}\$
13 & : \text{rab}\$
14 & : \text{ab}\$
15 & : \text{b}\$
16 & : \$
\end{align*}
BWT - Example

\[ T = \text{abracadabracarab}\$

Suffix Array

<table>
<thead>
<tr>
<th>Index</th>
<th>Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>$</td>
</tr>
<tr>
<td>14</td>
<td>ab$</td>
</tr>
<tr>
<td>10</td>
<td>acarab$</td>
</tr>
<tr>
<td>12</td>
<td>arab$</td>
</tr>
<tr>
<td>15</td>
<td>b$</td>
</tr>
<tr>
<td>1</td>
<td>bracadabracarab$</td>
</tr>
<tr>
<td>8</td>
<td>bracarab$</td>
</tr>
<tr>
<td>4</td>
<td>cadabracarab$</td>
</tr>
<tr>
<td>11</td>
<td>carab$</td>
</tr>
<tr>
<td>6</td>
<td>dabracarab$</td>
</tr>
<tr>
<td>13</td>
<td>rab$</td>
</tr>
<tr>
<td>2</td>
<td>racadabracarab$</td>
</tr>
<tr>
<td>9</td>
<td>racarab$</td>
</tr>
</tbody>
</table>
BWT - Example

\[ T = \text{abracadabracarab}$\]

| 16 | $ | \text{b} |
| 14 | ab$ | \text{r} |
| 0  | abracadabracarab$ | \text{d} |
| 7  | abracarab$ | \text{r} |
| 3  | acadabracarab$ | \text{r} |
| 10 | acarab$ | \text{r} |
| 5  | adabracarab$ | \text{c} |
| 12 | arab$ | \text{c} |
| 15 | b$ | \text{a} |
| 1  | bracadabracarab$ | \text{a} |
| 8  | bracarab$ | \text{a} |
| 4  | cadabracarab$ | \text{a} |
| 11 | carab$ | \text{a} |
| 6  | dabracarab$ | \text{a} |
| 13 | rab$ | \text{a} |
| 2  | racadabracarab$ | \text{b} |
| 9  | racarab$ | \text{b} |
BWT - Example

\[ T = \text{abracadabracarab$} \]

\[
\begin{array}{ll}
\$ & b \\
a & r \\
a & \$
\end{array}
\]

\[
\begin{array}{ll}
a & d \\
a & r \\
a & r \\
a & c \\
a & c \\
b & r \\
b & a \\
b & a \\
\end{array}
\]

\[
\begin{array}{ll}
c & a \\
c & a \\
d & a \\
r & a \\
r & a \\
r & b \\
r & b \\
\end{array}
\]

BWT
BWT - Reconstructing $T$ from BWT

$$T = \begin{array}{c}
\text{b} \\
\text{r} \\
\$ \\
\text{d} \\
\text{r} \\
\text{r} \\
\text{c} \\
\text{c} \\
\text{a} \\
\text{a} \\
\text{a} \\
\text{a} \\
\text{a} \\
\text{b} \\
\text{b}
\end{array}$$
BWT - Reconstructing $T$ from BWT

$$T =$$

1. Sort BWT to retrieve first column $F$
### BWT - Reconstructing T from BWT

<table>
<thead>
<tr>
<th>( T ) =</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
</tr>
<tr>
<td>6</td>
<td>a</td>
</tr>
<tr>
<td>7</td>
<td>a</td>
</tr>
<tr>
<td>8</td>
<td>b</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
</tr>
<tr>
<td>10</td>
<td>b</td>
</tr>
<tr>
<td>11</td>
<td>c</td>
</tr>
<tr>
<td>12</td>
<td>c</td>
</tr>
<tr>
<td>13</td>
<td>d</td>
</tr>
<tr>
<td>14</td>
<td>r</td>
</tr>
<tr>
<td>15</td>
<td>r</td>
</tr>
<tr>
<td>16</td>
<td>r</td>
</tr>
</tbody>
</table>

2. Find last symbol $ in \( F \) at position 0 and write to output
## BWT - Reconstructing T from BWT

The BWT (Burrows-Wheeler Transform) is a reversible transformation of a string. Given a string $T$, its BWT is a permutation of its characters that is useful in text compression and indexing.

1. **Symbol preceding $\$ \text{ in } T \text{ is } BWT[0] = b$.**
2. **Write to output**

### Example

<table>
<thead>
<tr>
<th>T</th>
<th>b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
</tr>
<tr>
<td>6</td>
<td>a</td>
</tr>
<tr>
<td>7</td>
<td>a</td>
</tr>
<tr>
<td>8</td>
<td>b</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
</tr>
<tr>
<td>10</td>
<td>b</td>
</tr>
<tr>
<td>11</td>
<td>c</td>
</tr>
<tr>
<td>12</td>
<td>c</td>
</tr>
<tr>
<td>13</td>
<td>d</td>
</tr>
<tr>
<td>14</td>
<td>r</td>
</tr>
<tr>
<td>15</td>
<td>r</td>
</tr>
<tr>
<td>16</td>
<td>r</td>
</tr>
</tbody>
</table>
### BWT - Reconstructing $T$ from BWT

<table>
<thead>
<tr>
<th>T</th>
<th>b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
</tr>
<tr>
<td>6</td>
<td>a</td>
</tr>
<tr>
<td>7</td>
<td>a</td>
</tr>
<tr>
<td>8</td>
<td>b</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
</tr>
<tr>
<td>10</td>
<td>b</td>
</tr>
<tr>
<td>11</td>
<td>c</td>
</tr>
<tr>
<td>12</td>
<td>c</td>
</tr>
<tr>
<td>13</td>
<td>d</td>
</tr>
<tr>
<td>14</td>
<td>r</td>
</tr>
<tr>
<td>15</td>
<td>r</td>
</tr>
<tr>
<td>16</td>
<td>r</td>
</tr>
</tbody>
</table>

3. As there are no $b$ before $BWT[0]$, we know that this $b$ corresponds to the first $b$ in $F$ at pos $F[8]$. 
BWT - Reconstructing $T$ from BWT

$$T = \text{ab}$

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### BWT - Reconstructing $T$ from BWT

<table>
<thead>
<tr>
<th>$T$</th>
<th>ab$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
</tr>
<tr>
<td>6</td>
<td>a</td>
</tr>
<tr>
<td>7</td>
<td>a</td>
</tr>
<tr>
<td>8</td>
<td>b</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
</tr>
<tr>
<td>10</td>
<td>b</td>
</tr>
<tr>
<td>11</td>
<td>c</td>
</tr>
<tr>
<td>12</td>
<td>c</td>
</tr>
<tr>
<td>13</td>
<td>d</td>
</tr>
<tr>
<td>14</td>
<td>r</td>
</tr>
<tr>
<td>15</td>
<td>r</td>
</tr>
<tr>
<td>16</td>
<td>r</td>
</tr>
</tbody>
</table>

BWT - Reconstructing T from BWT

\[ T = \text{rab$} \]

\[ \begin{array}{c|c|c}
0 & $ & b \\
1 & a & r \\
2 & a & $ \\
3 & a & d \\
4 & a & r \\
5 & a & r \\
6 & a & c \\
7 & a & c \\
8 & b & a \\
9 & b & a \\
10 & b & a \\
11 & c & a \\
12 & c & a \\
13 & d & a \\
14 & r & a \\
15 & r & b \\
16 & r & b \\
\end{array} \]

6. Output \( BWT[1] = r \) and map \( r \) to \( F[14] \)
BWT - Reconstructing T from BWT

\[ T = \text{arab}\$ \]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>a</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>b</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>b</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>c</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>c</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>d</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>r</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>r</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>r</td>
<td>b</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7. Output
\[ BWT[14] = a \]
and map \( a \) to \( F[7] \)
### BWT - Reconstructing $T$ from BWT

Let $T = \text{arab}$.

<table>
<thead>
<tr>
<th>Index</th>
<th>Symbol</th>
<th>$T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$</td>
<td>$b</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>$r</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>$d</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>$r</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>$r</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>$c</td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>$c</td>
</tr>
<tr>
<td>7</td>
<td>a</td>
<td>$c</td>
</tr>
<tr>
<td>8</td>
<td>b</td>
<td>$a</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>$a</td>
</tr>
<tr>
<td>10</td>
<td>b</td>
<td>$a</td>
</tr>
<tr>
<td>11</td>
<td>c</td>
<td>$a</td>
</tr>
<tr>
<td>12</td>
<td>c</td>
<td>$a</td>
</tr>
<tr>
<td>13</td>
<td>d</td>
<td>$a</td>
</tr>
<tr>
<td>14</td>
<td>r</td>
<td>$a</td>
</tr>
<tr>
<td>15</td>
<td>r</td>
<td>$b</td>
</tr>
<tr>
<td>16</td>
<td>r</td>
<td>$b</td>
</tr>
</tbody>
</table>

# BWT - Reconstructing $T$ from BWT

Let $T = \text{arab}$

<table>
<thead>
<tr>
<th></th>
<th>$T$</th>
<th>arab$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$$</td>
<td>$b$</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>$$</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>7</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>8</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>10</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>11</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>12</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>13</td>
<td>d</td>
<td>a</td>
</tr>
<tr>
<td>14</td>
<td>r</td>
<td>a</td>
</tr>
<tr>
<td>15</td>
<td>r</td>
<td>b</td>
</tr>
<tr>
<td>16</td>
<td>r</td>
<td>b</td>
</tr>
</tbody>
</table>


---

**BWT** - Reconstructing $T$ from BWT

<table>
<thead>
<tr>
<th></th>
<th>$T$</th>
<th>arab$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$$</td>
<td>$b$</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>$$</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>7</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>8</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>10</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>11</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>12</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>13</td>
<td>d</td>
<td>a</td>
</tr>
<tr>
<td>14</td>
<td>r</td>
<td>a</td>
</tr>
<tr>
<td>15</td>
<td>r</td>
<td>b</td>
</tr>
<tr>
<td>16</td>
<td>r</td>
<td>b</td>
</tr>
</tbody>
</table>
Thus, among the suffixes starting with $a$, the one preceding $SA[14]$ must be the last one.
BWT - Reconstructing T from BWT

\[ T = \text{abracadabracarab$} \]

<table>
<thead>
<tr>
<th>( i )</th>
<th>( T[i] )</th>
<th>( BWT[i] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>$</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>7</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>8</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>10</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>11</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>12</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>13</td>
<td>d</td>
<td>a</td>
</tr>
<tr>
<td>14</td>
<td>r</td>
<td>a</td>
</tr>
<tr>
<td>15</td>
<td>r</td>
<td>b</td>
</tr>
<tr>
<td>16</td>
<td>r</td>
<td>b</td>
</tr>
</tbody>
</table>
## Searching using the BWT

Given:

- $T = \text{abracadabra}c\text{br}\$,
- $P = \text{abr}$

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$ $</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>$ $</td>
<td>d</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>d</td>
<td>r</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>r</td>
<td>r</td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>7</td>
<td>a</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>b</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>10</td>
<td>b</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>11</td>
<td>c</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>12</td>
<td>c</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>13</td>
<td>d</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>14</td>
<td>r</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>15</td>
<td>r</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>r</td>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>
Searching using the BWT

\[ T = \text{abracadabracarab}$, \quad P = \text{abr} \]

Search backwards, start by finding the \( r \) interval in \( F \)
Searching using the BWT

$T = abracadabra$abra$, P = abr$

<table>
<thead>
<tr>
<th>0</th>
<th>$</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>$</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>r</td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>7</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>8</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>10</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>11</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>12</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>13</td>
<td>d</td>
<td>a</td>
</tr>
<tr>
<td>14</td>
<td>r</td>
<td>a</td>
</tr>
<tr>
<td>15</td>
<td>r</td>
<td>b</td>
</tr>
<tr>
<td>16</td>
<td>r</td>
<td>b</td>
</tr>
</tbody>
</table>

Search backwards, start by finding the $r$ interval in $F$.
Searching using the BWT

\[ T = \text{abracadabracarab}$, \quad P = \text{ab}r \]

How many \( b \)'s are the \( r \) interval in \( BWT[14, 16] \) ? 2
### Searching using the BWT

\[ T = \text{abracadabra$c$}, \quad P = abr \]

<table>
<thead>
<tr>
<th>BWT</th>
<th>Wavelet Trees</th>
<th>CSA Usage</th>
<th>Compressed Suffix Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>r</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How many suffixes starting with \( b \) are smaller than those \( 2 \)? 1 at \( BWT[0] \)
Searching using the BWT

\[ T = \text{abracadabra}$, \ P = \text{abr} \]

Thus, all suffixes starting with \textit{br} are in \(SA[9, 10]\).
Searching using the BWT

\[ T = \text{abracadabracarab}\$, \ P = \text{abr} \]

How many of the suffixes starting with \textit{br} are preceded by \textit{a}? 2
### Searching using the BWT

Given:

- $T = \text{abracadabracarab}$
- $P = \text{abr}$

#### BWT Table:

<table>
<thead>
<tr>
<th>0</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
</tr>
<tr>
<td>6</td>
<td>a</td>
</tr>
<tr>
<td>7</td>
<td>a</td>
</tr>
<tr>
<td>8</td>
<td>b</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
</tr>
<tr>
<td>10</td>
<td>b</td>
</tr>
<tr>
<td>11</td>
<td>c</td>
</tr>
<tr>
<td>12</td>
<td>c</td>
</tr>
<tr>
<td>13</td>
<td>d</td>
</tr>
<tr>
<td>14</td>
<td>r</td>
</tr>
<tr>
<td>15</td>
<td>r</td>
</tr>
<tr>
<td>16</td>
<td>r</td>
</tr>
</tbody>
</table>

**Question:** How many of the suffixes smaller than $br$ are preceded by $a$?

**Answer:** 1
Searching using the BWT

\[ T = \text{abracadabra} \text{br} \text{c} \text{a} \text{d} \text{a} \text{r} \text{b}\$,  \ P = \text{abr} \]

There are 2 occurrences of \text{abr} in \( T \) corresponding to suffixes \( SA[2, 3] \)
Searching using the BWT

- We only require $F$ and $BWT$ to search and recover $T$

- We only had to count the number of times a symbol $s$ occurs within an interval, and before that interval $BWT[i, j]$

- Equivalent to $Rank_s(BWT, i)$ and $Rank_s(BWT, j)$

- Need to perform $Rank$ on non-binary alphabets efficiently
Wavelet Trees - Overview

- Data structure to perform \textit{Rank} and \textit{Select} on non-binary alphabets of size $\sigma$ in $O(\log_2 \sigma)$ time

- Decompose non-binary \textit{Rank} operations into binary \textit{Rank}’s via tree decomposition

- Space usage $n \log \sigma + o(n \log \sigma)$ bits. Same as original sequence + Rank + Select overhead
Wavelet Trees - Example

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>00</td>
</tr>
<tr>
<td>a</td>
<td>010</td>
</tr>
<tr>
<td>b</td>
<td>011</td>
</tr>
<tr>
<td>c</td>
<td>10</td>
</tr>
<tr>
<td>d</td>
<td>110</td>
</tr>
<tr>
<td>r</td>
<td>111</td>
</tr>
</tbody>
</table>

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
b r $ d r r c c a a a a a a a a b b

Symbol Codeword
Wavelet Trees - Example

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
b r $ d r r c c a a a a a a a b b
0 1 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0
```
Wavelet Trees - Example

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
b r $ d r r c c a a a a a a a b b
0 1 0 1 1 1 1 0 0 0 0 0 0 0 0 0
Wavelet Trees - Example

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
b r $ d r r c c a a a a a a b b
0 1 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0

0 1 2 3 4 5 6 7 8 9 10
b $ a a a a a a a a b b
1 0 1 1 1 1 1 1 1 1 1 1
Wavelet Trees - Example

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
b r $ d r r c c a a a a a a a b b
0 1 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0

0 1 2 3 4 5 6 7 8 9 10
b $ a a a a a a a a b b
1 0 1 1 1 1 1 1 1 1 1
```

```
0 1 2 3 4 5
r d r r c c
1 1 1 1 0 0
```
Wavelet Trees - Example

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
b r $ d r r c c a a a a a a a a a b b
0 1 0 1 1 1 1 0 0 0 0 0 0 0 0 0
```

```
0 1 2 3 4 5 6 7 8 9 10
b $ a a a a a a a a b b
1 0 1 1 1 1 1 1 1 1
```

```
0 1 2 3 4 5
r d r r c c
c
1 1 1 1 0 0
```

```
$
```
Wavelet Trees - Example

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
b r $ d r r c c a a a a a a a b b
0 1 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0
```

```
0 1 2 3 4 5 6 7 8 9 10
b $ a a a a a a a a a a b b
1 0 1 1 1 1 1 1 1 1 1 1
```

```
0 1 2 3 4 5
r d r r c c
1 1 1 1 1 0 0
```

```
0 1 2 3 4 5 6 7 8 9
b a a a a a a a a a a b b
1 0 0 0 0 0 0 0 0 1 1
```
Wavelet Trees - Example

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
b r $ d r r c c a a a a a a a b b
0 1 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0
```

```
0 1 2 3 4 5 6 7 8 9 10
b $ a a a a a a a a a a b b
1 0 1 1 1 1 1 1 1 1 1 1
```

```
0 1 2 3 4 5 6 7 8 9
$ b a a a a a a a a a a b b
1 0 0 0 0 0 0 0 0 1 1
```

```
0 1 2 3 4 5
r d r r c c
1 1 1 1 0 0
```
Wavelet Trees - Example
Wavelet Trees - Example

The diagram illustrates a wavelet tree for a string "brdrdrcacaaabaabbb010111110000000000". The tree is constructed by dividing the string into blocks and using bit vectors to represent the positions of each character within the suffixes of the string. The tree structure is shown through a binary tree representation, where each node corresponds to a bit vector indicating the presence or absence of a character at each position in the suffixes. The tree efficiently supports string matching and other operations on the data.
Wavelet Trees - Example

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
b r $ d r r c c a a a a a a a b b
0 1 0 1 1 1 1 1 0 0 0 0 0 0 0 0

0 1 2 3 4 5 6 7 8 9 10
b $ a a a a a a a a b b
1 0 1 1 1 1 1 1 1 1 1

0 1 2 3 4 5
r d r r c c
1 1 1 1 1 0 0

$ 0 1 2 3 4 5 6 7 8 9
b a a a a a a a a b b
1 0 0 0 0 0 0 0 0 1 1

a 0 1 2 3
r d r r
1 0 1 1

b c

d r
Wavelet Trees - What is actually stored

0 1 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0

1 0 1 1 1 1 1 1 1 1

$ 1 0 0 0 0 0 0 0 0 1 1

1 1 1 1 1 0 0

a b c d r
Wavelet Trees - Performing $\text{Rank}_a(BWT, 11)$

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
b & r & $ & d & r & r & c & c & a & a & a & a & a & a & b & b \\
0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
b & $ & a & a & a & a & a & a & a & a & b & b \\
0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
r & d & r & r & c & c \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
$ \\
0 & 1 & 2 & 3 & 4 & 5 \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
a \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
b \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
c \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
d \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
r \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]
Wavelet Trees - Performing $\text{Rank}_a(BWT, 11)$

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
b r $ d r r c c a a a a a a a a a a
0 1 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0
```

```
0 1 2 3 4 5 6 7 8 9 10
b $ a a a a a a a a a a a a a a
1 0 1 1 1 1 1 1 1 1 1 1 1
```

```
0 1 2 3 4 5
r d r r c c c
1 1 1 1 0 0
```

$\text{Rank}_a(BWT, 11)$ is performed on the wavelet tree. The value for the character $a$ at position 11 is 5, indicating that there are 5 characters before position 11 that are equal to $a$.
Wavelet Trees - Performing $\text{Rank}_a(BWT, 11)$

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

$\text{br}\$ $d$ $r$ $c$ $c$ $a$ $a$ $a$ $a$

0 1 0 1 1 1 1 1 0 0 0 0 0 0 0 0

$0 1 2 3 4 5$
$\text{rdrrcc}$
$1 1 1 1 0 0$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$

$0 1 2 3 4 5 6 7 8 9$
$\text{b a a a a a a a a b b}$
$1 0 0 0 0 0 0 0 0 1 1$
Wavelet Trees - Performing $\text{Rank}_a(BWT, 11)$

```
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
b r $ d r r c c a a a a a a a b b
0 1 0 1 1 1 1 1 0 0 0 0 0 0 0 0
```

```
0 1 2 3 4 5 6 7 8 9 10
b $ a a a a a a a a b b
1 0 1 1 1 1 1 1 1 1 1 1
```

```
0 1 2 3 4 5
r d r r c c
1 1 1 1 1 0 0
```

```
$ 0 1 2 3 4 5 6 7 8 9
b a a a a a a a a a b b
1 0 0 0 0 0 0 0 0 1 1
```

```
0 1 2 3 4 5
r d r r r
1 0 1 1
```

```
c 0 1 2 3
```

```
d 0 1 2 3
```

```
r 0 1 2 3
```
Wavelet Trees - Performing $\text{Rank}_a(BWT, 11)$
Wavelet Trees - Performing $\text{Rank}_a(BWT, 11)$
Wavelet Trees - Performing $\text{Rank}_a(BWT, 11)$

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
\hline
\text{b r d r c c a a a a} & \text{a a a b b} \\
0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Wavelet Trees - Space Usage

Currently: \( n \log \sigma + o(n \log \sigma) \) bits. Still larger than the original text!

How can we do better?

- Compressed bitvectors
Currently: $n \log \sigma + o(n \log \sigma)$ bits. Still larger than the original text!

How can we do better?

- Picking the codewords for each symbol smarter!
Wavelet Trees - Space Usage

Currently

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Freq</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>a</td>
<td>7</td>
<td>010</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td>110</td>
</tr>
<tr>
<td>r</td>
<td>3</td>
<td>111</td>
</tr>
</tbody>
</table>

Bits per symbol: 2.82

Huffman Shape:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Freq</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>1</td>
<td>1100</td>
</tr>
<tr>
<td>a</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>101</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>111</td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td>1101</td>
</tr>
<tr>
<td>r</td>
<td>3</td>
<td>100</td>
</tr>
</tbody>
</table>

Bits per symbol: 2.29

Space usage of Huffman shaped wavelet tree:
\[ H_0(T)n + o(H_0(T)n) \] bits.

Even better: Huffman shape + compressed bitvectors
CSA-WT - Space Usage in practice

Index size [% of original text size]

- dna.200MB
- proteins.200MB
- dblp.xml.200MB
- english.200MB

Index time per character [ns]

- CSA
- CSA++
- CSA-SADA
- CSA-OPF
- FM-HF-BVIL
- FM-HF-RRR
- FM-FB-BVIL
- FM-FB-HYB

Count time per character [ns]

- CSA
- CSA++
- CSA-SADA
- CSA-OPF
- FM-HF-BVIL
- FM-HF-RRR
- FM-FB-BVIL
- FM-FB-HYB
#include "sdsl/suffix_arrays.hpp"
#include "sdsl/bit_vectors.hpp"
#include "sdsl/wavelet_trees.hpp"

int main(int argc, char** argv) {
    std::string input_file = argv[1];
    // use a compressed bitvector
    using bv_type = sdsl::hyb_vector<>;
    // use a huffman shaped wavelet tree
    using wt_type = sdsl::wt_huff<bv_type>;
    // use a wt based CSA
    using csa_type = sdsl::csa_wt<wt_type>;
    csa_type csa;
    sdsl::construct(csa, input_file, 1);
    sdsl::store_to_file(csa, out_file);
}
CSA-WT - Trade-offs in SDSL

```cpp
// use a regular bitvector
using bv_type = sdsl::bit_vector;
// 5% overhead rank structure
using rank_type = sdsl::rank_support_v5<1>;
// don't need select so we just use
// scanning which is O(n)
using select1_type = sdsl::select_support_scan<1>;
using select0_type = sdsl::select_support_scan<0>;
// use a huffman shaped wavelet tree
using wt_type = sdsl::wt_huff<bv_type,
                        rank_type,
                        select1_type,
                        select0_type>;
using csa_type = sdsl::csa_wt<wt_type>;
csa_type csa;
sdsl::construct(csa, input_file, 1);
sdsl::store_to_file(csa, out_file);
```
```cpp
int main(int argc, char** argv) {
    std::string input_file = argv[1];
    sds::csa_wt<> csa;
    sds::construct(csa, input_file, 1);

    std::string pattern = "abr";
    auto nocc = sds::count(csa, pattern);
    auto occs = sds::locate(csa, pattern);
    for (auto& occ : occs) {
        std::cout << "found at pos " << occ << std::endl;
    }

    auto snippet = sds::extract(csa, 5, 12);
    std::cout << "snippet = " << snippet << std::endl;
}
```
sds loosen::csa_wt<> csa; //
sds loosen::construct(csa, "this-file.cpp", 1);
std loosen::cout << "count("""") : "
    << sds loosen::count(csa, "") << endl;
auto occs = sds loosen::locate(csa, "\n");
sort(occs.begin(), occs.end());
auto max_line_length = occs[0];
for (size_t i=1; i < occs.size(); ++i)
    max_line_length = std loosen::max(max_line_length,
                                    occs[i]-occs[i-1]+1);
std loosen::cout << "max line length : "
    << max_line_length << endl;
CSA-WT - Searching - Words

32 bit integer words:

```cpp
sdsl::csa_wt_int<> csa;
// file containing uint32_t ints
sdsl::construct(csa, "words.u32", 5);
std::vector<uint32_t> pattern = {532432, 43433};
std::cout << "count() : "
   << sdsl::count(csa, pattern) << endl;
```

$\log_2 \sigma$ bit words in SDSL format:

```cpp
sdsl::csa_wt_int<> csa;
// file containing a serialized sdsl::int_vector ints
sdsl::construct(csa, "words.sdsl", 0);
std::vector<uint32_t> pattern = {532432, 43433};
std::cout << "count() : "
   << sdsl::count(csa, pattern) << endl;
```
CSA - Usage Resources

Tutorial: http://simongog.github.io/assets/data/sdsl-slides/tutorial

Cheatsheet: http://simongog.github.io/assets/data/sdsl-cheatsheet.pdf

Examples: https://github.com/simongog/sdsl-lite/examples

Tests: https://github.com/simongog/sdsl-lite/test
Compressed Suffix Trees

- Compressed representation of a Suffix Tree
- Internally uses a CSA
- Store extra information to represent tree shape and node depth information
- Three different CST types available in SDSL
Compressed Suffix Trees - CST

- Use a succinct tree representation to store suffix tree shape

- Compress the LCP array to store node depth information

Operations:
root, parent, first_child, iterators, sibling, depth, node_depth, edge, children... many more!
CST - Example

1 using csa_type = sdsl::csa_wt<>;
2 sdsl::cst_sct3<csa_type> cst;
3 sdsl::construct_im(cst, "ananas", 1);
4 for (auto v : cst) {
5     cout << cst.depth(v) << "-" << cst.lb(v) << " ","
6     << cst.rb(v) << " ]" << endl;
7 }
8 auto v = cst.select_leaf(2);
9 for (auto it = cst.begin(v); it != cst.end(v); ++it) {
10    auto node = *it;
11    cout << cst.depth(v) << "-" << cst.lb(v) << " ","
12    << cst.rb(v) << " ]" << endl;
13 }
14 v = cst.parent(cst.select_leaf(4));
15 for (auto it = cst.begin(v); it != cst.end(v); ++it) {
16    cout << cst.depth(v) << "-" << cst.lb(v) << " ","
17    << cst.rb(v) << " ]" << endl;
18 }
CST - Space Usage Visualization

http://simongog.github.io/assets/data/space-vis.html
<table>
<thead>
<tr>
<th></th>
<th>Applications to NLP</th>
<th>LM fundamentals</th>
<th>LM complexity</th>
<th>LMs meet SA/ST</th>
<th>Query and construct</th>
<th>Experiments</th>
<th>Other Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Applications to NLP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>LM fundamentals</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>LM complexity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>LMs meet SA/ST</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Query and construct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Experiments</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Other Apps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Application to NLP: language modelling

1. Applications to NLP
2. LM fundamentals
3. LM complexity
4. LMs meet SA/ST
5. Query and construct
6. Experiments
7. Other Apps
Language models & succinct data structures

Count-based language models:

\[ P(w_i|w_1, \ldots, w_{i-1}) \approx P^{(k)}(w_i|w_{i-k}, \ldots, w_{i-1}) \]

Estimation from \( k \)-gram corpus statistics using ST/SA

- based arounds suffix arrays \([Zhang and Vogel, 2006]\)
- and suffix trees \([Kennington et al., 2012]\)
- practical using CSA/CST \([Shareghi et al., 2016b]\)

In all cases, on-the-fly calculation and no cap on \( k \) required.\(^1\)

Related, machine translation

Lookup of (dis)contiguous ‘phrases’, as part of dynamic phrase-table \([Callison-Burch et al., 2005, Lopez, 2008]\).

\(^1\)Caps needed on smoothing parameters \([Shareghi et al., 2016a]\).
Commonly, store probabilities for $k$-grams explicitly.

**Efficient storage**

- tries and hash tables for fast lookup [Heafield, 2011]
- lossy data structures [Talbot and Osborne, 2007]
- storage of approximate probabilities using quantisation and pruning [Pauls and Klein, 2011]
- parallel ‘distributed’ algorithms [Brants et al., 2007]

Overall: fast, but limited to fixed $m$-gram, and intensive hardware requirements.
Language models

Definition

A language model defines probability $P(w_i|w_1, \ldots, w_{i-1})$, often with a Markov assumption, i.e., $P \approx P^{(k)}(w_i|w_{i-k}, \ldots, w_{i-1})$.

Example: MLE for $k$-gram LM

$$P^{(k)}(w_i|w_{i-1}^{i-k}) = \frac{c(w_{i-k}^i)}{c(w_{i-k}^{i-1})}$$

- using count of context, $c(w_{i-k}^{i-1})$; and
- count of full $k$-gram, $c(w_{i-k}^{i})$

Notation: $w_{i}^{j} \triangleq (w_i, w_{i+1}, \ldots, w_j)$
Smoothed count-based language models

Interpolate or backoff from higher to lower order models

\[ P^{(k)}(w_i|w_{i-k}^{i-1}) = f(w_{i-k}^i) + g(w_{i-k}^{i-1})P^{(k-1)}(w_i|w_{i-k+1}^{i-1}) \]

terminating at unigram MLE, \( P^{(1)} \).

Selecting \( f \) and \( g \) functions

**Interpolation** \( f \) is a discounted function of the context and \( k \)-gram counts, reserving some mass for \( g \)

**Backoff** only one of \( f \) or \( g \) term is non-zero, based on whether full pattern is found

Involved computation of either the discount or normalisation.
Kneser-Ney smoothing (Kneser and Ney, 1995; Chen and Goodman, 1998)

### Intuition

Not all $k$-grams should be treated equally $\Rightarrow$ $k$-grams occurring in fewer contexts should carry lower weight.

### Example

*Fransisco* is a common unigram, but only occurs in one context, *San Franscisco*

Treat unigram *Fransisco* as having count 1.

Enacted through formulation based occurrence counts for scoring component $k < m$ grams and discount smoothing.
Kneser-Ney smoothing (Kneser and Ney, 1995; Chen and Goodman, 1998)

\[ P^{(k)}(w_i | w_{i-k}^{i-1}) = f(w_{i-k}^i) + g(w_{i-k}^i)P^{(k-1)}(w_i | w_{i-k+1}^{i-1}) \]

**Highest order** \( k = m \)

\[
\begin{align*}
f(w_{i-k}^i) &= \frac{[c(w_{i-k+1}^i) - D_k]^+}{c(w_{i-k+1}^{i-1})} \\
g(w_{i-k}^{i-1}) &= \frac{D_kN_{1+}(w_{i-k-1}^{i-1} \cdot)}{c(w_{i-k+1}^{i-1})}
\end{align*}
\]

\( 0 \leq D_k < 1 \) are discount constants.

**Lower orders** \( k < m \)

\[
\begin{align*}
f(w_{i-k}^i) &= \frac{[N_{1+}(w_{i-k+1}^i) - D_k]^+}{N_{1+}(w_{i-k-1}^{i-1} \cdot)} \\
g(w_{i-k}^{i-1}) &= \frac{D_kN_{1+}(w_{i-k+1}^{i-1} \cdot)}{N_{1+}(w_{i-k-1}^{i-1} \cdot)}
\end{align*}
\]

Uses unique context counts, rather than counts directly.
Modified Kneser Ney

Discount component now a function of the $k$-gram count / occurrence count

$$D_k : [0, 1, 2, 3+] \rightarrow \mathcal{R}$$

Consequence: complication to $g$ term!

Now must incorporate the number of $k$-grams with given prefix

- with count 1, $N_1(w_{i-k+1}^{i-1} \cdot)$;
- with count 2, $N_2(w_{i-k+1}^{i-1} \cdot)$; and
- with count 3 or greater, $N_{1+} - N_1 - N_2$. 
Sufficient Statistics

Kneser Ney probability computation requires the following:

\[
\begin{align*}
&c(w^j_i) & \text{basic counts} \\
&N_{1+}(w^j_i \cdot) \\
&N_{1+}(\cdot w^j_i) \\
&N_{1+}(\cdot w^j_i \cdot) \\
&N_1(w^j_i \cdot) \\
&N_2(w^j_i \cdot)
\end{align*}
\]

\[
\begin{align*}
&\text{occurrence counts}
\end{align*}
\]

Other smoothing methods also require forms of occurrence counts, e.g., Good-Turing, Witten-Bell.
Construction and querying

Probabilities computed ahead of time

- Calculate a static hashtable or trie mapping $k$-grams to their probability and backoff values.
- **Big**: number of possible & observed $k$-grams grows with $k$

Querying

Lookup the longest matching span including the current token, and without the token. Probability computed from the full score and context backoff.
Query cost

German Europarl, KenLM trie
Cost of construction  German Europarl, KenLM trie

<table>
<thead>
<tr>
<th>MiB</th>
<th>secs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>750</td>
<td>2</td>
</tr>
<tr>
<td>1,500</td>
<td>3</td>
</tr>
<tr>
<td>2,250</td>
<td>4</td>
</tr>
<tr>
<td>2,850</td>
<td>5</td>
</tr>
<tr>
<td>3,350</td>
<td>6</td>
</tr>
<tr>
<td>3,750</td>
<td>7</td>
</tr>
<tr>
<td>4,150</td>
<td>8</td>
</tr>
<tr>
<td>4,550</td>
<td>9</td>
</tr>
<tr>
<td>4,950</td>
<td>10</td>
</tr>
</tbody>
</table>

- Memory: CoT (MiB)
- Time: Construction time (secs)

Note: The graph shows the incremental cost of construction with German Europarl data and KenLM trie.
Precomputing versus on-the-fly

Precomputing approach

- Does not scale gracefully to high order $m$;
- Large training corpora also problematic

Can be computed directly from a CST

- CST captures unlimited order $k$-grams (no limit on $m$);
- Many (but not all) statistics cheap to retrieve
- LM probabilities computed on-the-fly
## Sufficient statistics captured in suffix structures

Given the string $T = \text{abracadabra}$,$\text{bracarab}$, the suffix array $SA_i$ and the suffix array $T_{SA_i}$ are constructed as follows:

<table>
<thead>
<tr>
<th>$i$</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>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SA_i$</td>
<td>16</td>
<td>14</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>10</td>
<td>5</td>
<td>12</td>
<td>15</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>11</td>
<td>6</td>
<td>13</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>$T_{SA_i}$</td>
<td>$\text{$a\ a\ a\ a\ a\ a\ a\ b\ b\ b\ c\ c\ d\ r\ r\ r}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_{SA_{i-1}}$</td>
<td>$b\ r\ $\ d\ r\ r\ c\ c\ a\ a\ a\ a\ a\ a\ b\ b$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sufficient statistics captured in suffix structures

\[ T = \text{abracadabracarab}$ \]

<table>
<thead>
<tr>
<th>( i )</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>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>( SA_i )</td>
<td>16</td>
<td>14</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>10</td>
<td>5</td>
<td>12</td>
<td>15</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>11</td>
<td>6</td>
<td>13</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>( T_{SA_i} )</td>
<td>$</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>c</td>
<td>c</td>
<td>d</td>
<td>r</td>
<td>r</td>
</tr>
<tr>
<td>( T_{SA_{i-1}} )</td>
<td>b</td>
<td>r</td>
<td>$</td>
<td>d</td>
<td>r</td>
<td>r</td>
<td>c</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>
Sufficient statistics captured in suffix structures

\[ T = \texttt{abracadabracarab$} \]

\begin{array}{cccccccccccccccc}
 i & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
 SA_i & 16 & 14 & 0 & 7 & 3 & 10 & 5 & 12 & 15 & 1 & 8 & 4 & 11 & 6 & 13 & 2 & 9 \\
 T_{SA_i} & $ & a & a & a & a & a & a & b & b & b & c & c & d & r & r & r \\
 T_{SA_{i-1}} & b & r & $ & d & r & r & c & c & a & a & a & a & a & a & a & b & b \\
\end{array}

- \( c(\text{abra}) = 2 \) from CSA
  - range between \( lb = 3 \) and \( rb = 4 \), inclusive
Sufficient statistics captured in suffix structures

\[ T = \text{abracadabra} \text{carab}\$ \]

<table>
<thead>
<tr>
<th></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>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i)</td>
<td>16</td>
<td>14</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>10</td>
<td>5</td>
<td>12</td>
<td>15</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>11</td>
<td>6</td>
<td>13</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>(SA_i)</td>
<td>$</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>c</td>
<td>c</td>
<td>d</td>
<td>r</td>
<td>r</td>
<td>r</td>
</tr>
<tr>
<td>(T_{SA_i})</td>
<td>b</td>
<td>r</td>
<td>$</td>
<td>d</td>
<td>r</td>
<td>r</td>
<td>c</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>b</td>
</tr>
</tbody>
</table>

- \(c(\text{abra}) = 2\) from CSA
  - range between \(lb = 3\) and \(rb = 4\), inclusive
- \(N_{1+}(\cdot \text{abra}) = 2\) from BWT (wavelet tree)
  - size of set of preceding symbols \(\{\$, d\}\)
Occurrence counts from the suffix tree

Number of proceeding symbols, $N_{1+}(\alpha \cdot)$, is either
Occurrence counts from the suffix tree

Number of proceeding symbols, $N_{1+}(\alpha \bullet)$, is either

- 1 if internal to an edge (e.g., $\alpha = \text{abra}$)
Occurrence counts from the suffix tree

Number of proceeding symbols, $N_{1+}(\alpha \cdot)$, is either

- 1 if internal to an edge (e.g., $\alpha = \text{abra}$)
- degree($v$) otherwise (e.g., $\alpha = \text{ab}$ with degree 2)
More difficult occurrence counts

How to handle occurrence counts to both sides,

$$N_{1+}(\bullet \alpha \bullet) = \left| \{w\alpha v, \text{s.t. } c(w\alpha v) \geq 1\} \right|$$

and specific value $i$ occurrence counts,

$$N_i(\alpha \bullet) = \left| \{\alpha v, \text{s.t. } c(\alpha v) = i\} \right|$$

No simple mapping to CSA/CST algorithm

Iterative (costly!) solution used instead:

- enumerate extensions to one side
- accumulate counts (to the other side, or query if $c = i$)
Algorithm outline

Step 1: search for pattern

Backward search for each symbol, in right-to-left order. Results in bounds $[lb, rb]$ of matching patterns.

Step 2: find statistics

- **count** $c(a b r a) = rb - lb - 1$ (or 0 on failure.)
- **left occ.** $N_{1+}(\cdot w_i^j)$ can be computed from BWT (over preceding symbols.)
- **right occ.** $N_{1+}(w_i^j \cdot)$ based on shape of the *suffix tree.*
- **twin occ. etc** . . . increasingly complex . . .

Nb. illustrating ideas with basic SA/STs; in practice CSA/CSTs.
Step 2: Compute statistics

Given range $[lb, rb]$ for matching pattern, $\alpha$, can compute:

- count, $c(\alpha) = (rb - lb + 1)$
- occurrence count, $N_{1+}(\cdot \alpha) = \text{interval-symbols}(lb, rb)$

with time complexity

- $o(1)$; and

- $O(N_{1+}(\cdot \alpha) \cdot \log \sigma)$ where $\sigma$ is the size of the vocabulary

What about the other required occurrence counts?
Querying algorithm: one-shot

\[ P(\text{ham}) \]

- green
- eggs
- and
- ham
Querying algorithm: one-shot

\[ P(\text{ham}|\text{and}) \]

\[ P(\text{ham}) \]
Querying algorithm: one-shot

At each step: 1) extend search for context and full pattern; 2) compute $c$ and/or $N_{1+}$ counts.

$P(\text{ham}|\text{eggs and})$

$P(\text{ham}|\text{and})$

$P(\text{ham})$

---

green
eggs
and
ham
Querying algorithm: one-shot

\[ P(\text{ham} | \text{green eggs and}) \]

\[ P(\text{ham} | \text{eggs and}) \]

\[ P(\text{ham} | \text{and}) \]

\[ P(\text{ham}) \]
Querying algorithm: one-shot

\[ P(\text{ham}|\text{green eggs and}) \]

\[ P(\text{ham}|\text{eggs and}) \]

\[ P(\text{ham}|\text{and}) \]

\[ P(\text{ham}) \]

At each step: 1) extend search for context and full pattern; 2) compute \( c \) and/or \( N^{1+} \) counts.
Querying algorithm: full sentence

Reuse matches

Full matches in one step become context matches for next step. E.g., *green eggs and ham* ⇐ *green eggs and*

- recycle the CSA matches from previous query, halving search cost
- N.b., can’t recycle counts, as mostly use different types of occurrence counts on numerator cf denominator

Unlimited application

No bound on size of match, can continue until pattern unseen in training corpus.
Construction algorithm

1. Sort suffixes (on disk)
2. Construct CSA
3. Construct CST
4. Compute discounts
   - efficient using traversal of $k$-grams in the CST (up to a given depth)
5. Precompute some expensive values
   - again use traversal of $k$-grams in the CST
Accelerating expensive counts

Iterative calls, e.g., $N_{1+}(\cdot \alpha \cdot)$ account for majority of runtime.

Solution: cache common values

- store values for common entries, i.e., highest nodes in CST
- values are integers, mostly with low values $\rightarrow$ very compressable!

Technique

- store bit vector, $bv$, of length $n$, where $bv[i]$ records whether value for $i$ is cached
- store cached values in an integer vector, $v$, in linear order
- retrieve $i^{th}$ value using $v[\text{rank}_1(bv, i)]$
### Effect of caching

#### On-the-fly

- $N_{123+}(\alpha.)$
- $N_{123+}(\alpha.)$
- $N_1+.\alpha.$
- $N_1+.\alpha.$
- $N_1+.\alpha.$
- $N_1+.\alpha.$
- backward-search

#### Precomputed

- Precomputed m-gram

<table>
<thead>
<tr>
<th>m-gram</th>
<th>Time (sec)</th>
<th>Time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4ms</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>8ms</td>
<td>4ms</td>
</tr>
<tr>
<td>5</td>
<td>8ms</td>
<td>8ms</td>
</tr>
<tr>
<td>8</td>
<td>8ms</td>
<td>12ms</td>
</tr>
<tr>
<td>$\infty$</td>
<td>8ms</td>
<td>20ms</td>
</tr>
</tbody>
</table>

+15-20% space requirement ($\leq 10$-gram)
Timing versus other LMs: Small DE Europarl

![Graph showing the comparison of different language models on construction and load+query times versus memory usage.](image)

- **CST on-the-fly**
- **KenLM (trie)**
- **SRILM**
- **CST precompute**
- **KenLM (probing)**
**Timing versus other LMs: Large DE Commoncrawl**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>

**m**
- 2gram
- 3gram
- 4gram
- 5gram
- 8gram
- 10gram

**Method**
- ken (pop.)
- ken (lazy)
- cst
## Perplexity: usefulness of large or infinite context

<table>
<thead>
<tr>
<th>Training</th>
<th>size (M)</th>
<th>perplexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tokens</td>
<td>sents</td>
</tr>
<tr>
<td>Europarl</td>
<td>55</td>
<td>2.2</td>
</tr>
<tr>
<td>NCrawl2007</td>
<td>37</td>
<td>2.0</td>
</tr>
<tr>
<td>NCrawl2008</td>
<td>126</td>
<td>6.8</td>
</tr>
<tr>
<td>NCrawl2013</td>
<td>641</td>
<td>35.1</td>
</tr>
<tr>
<td>NCrawl2014</td>
<td>845</td>
<td>46.3</td>
</tr>
<tr>
<td>All combined</td>
<td>2560</td>
<td>139.3</td>
</tr>
<tr>
<td>CCrawl32G</td>
<td>5540</td>
<td>426.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>unit</th>
<th>time (s)</th>
<th>mem (GiB)</th>
<th>$m = 5$</th>
<th>$m = 10$</th>
<th>$m = 20$</th>
<th>$m = \infty$</th>
</tr>
</thead>
<tbody>
<tr>
<td>word</td>
<td>8164</td>
<td>6.29</td>
<td>73.45</td>
<td>68.66</td>
<td>68.76</td>
<td>68.80</td>
</tr>
<tr>
<td>byte</td>
<td>17935</td>
<td>18.58</td>
<td>3.93</td>
<td>2.69</td>
<td>2.37</td>
<td>2.33</td>
</tr>
</tbody>
</table>
Practical exercise

Finding concordances for an arbitrary $k$-gram pattern:

Outline

- find count of $k$-gram in large corpus
- show tokens to left or right, sorted by count
- find pairs of tokens occurring to left and right

Tools

- building a CSA and CST
- searching for pattern
- querying CST path label & children (to right)
- querying WT for symbols to left
Semi-External Indexes

Semi-External Suffix Array (RoSA)

- Store the ”top” part of a suffix tree in memory (using a compressed structure)
- If pattern short and frequent. Answer from in-memory structure (fast!)
- If pattern long or infrequent perform disk access
- Implemented, complicated, currently not used in practice
Range Minimum/Maximum Queries

- Given an array $A$ of $n$ items
- For any range $A[i, j]$ answer in constant time, what is the largest / smallest item in the range
- Space usage: $2n + o(n)$ bits. $A$ not required!
Compressed Tries / Dictionaries

- Support $\text{LOOKUP}(s)$ which returns unique id if string $s$ is in dict or $-1$ otherwise
- Support $\text{RETRIEVE}(i)$ return string with id $i$
- Very compact. 10% – 20% of original data
- Very fast lookup times
- Efficient construction
Graph Compression

Retrieving direct neighbors for page 10
Other applications

- Store the "top" part of a suffix tree in memory (using a compressed structure)
- If pattern short and frequent. Answer from in-memory structure (fast!)
- If pattern long or infrequent perform disk access
- Implemented, complicated, currently not used in practice
Basic succinct structures rely on bitvectors and operations \texttt{Rank} and \texttt{Select}

More complex structures are composed of these basic building blocks

Many trade-offs exist

Practical, highly engineered open source implementations exist and can be used within minutes in industry and academia

Other fields such as Information Retrieval, Bioinformatics have seen many papers using these succinct structures in recent years
Compact Data Structures,
A practical approach
Gonzalo Navarro
Cambridge University Press, 2016
Resources II

- Overview of compressed text indexes: [Ferragina et al., 2008, Navarro and Mäkinen, 2007]
- Bitvectors: [Gog and Petri, 2014]
- Document Retrieval: [Navarro, 2014a]
- Compressed Suffix Trees: [Sadakane, 2007, Ohlebusch et al., 2010]
- Wavelet Trees: [Navarro, 2014b]
- Compressed Tree Representations: [Navarro and Sadakane, 2016]


Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences.
*ACM Comp. Surv.*, 46(4.52).

Wavelet trees for all.

Compressed full-text indexes.

Compressed tree representations.

CST++.

In *Proceedings of the International Symposium on String Processing and Information Retrieval*.


Faster and smaller n-gram language models.

In *Proceedings of the Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*.


Compressed suffix trees with full functionality.


*Transactions of the Association for Computational Linguistics, 4:*477–490.
