Space Efficient Data Structures and FM index

Venkatesh Raman

The Institute of Mathematical Sciences, Chennai

NISER Bhubaneshwar, February 9, 2019

Overview

Overview

- Plan of the talk

Overview

- Plan of the talk
  - Why Space efficient?

Overview

- Plan of the talk
  - Why Space efficient?
  - What we mean by efficient?

Overview

- Plan of the talk
  - Why Space efficient?
  - What we mean by efficient? (information theory lower bound)

Overview

- Plan of the talk
  - Why Space efficient?
  - What we mean by efficient? (information theory lower bound)
  - How

Overview

- Plan of the talk
  - Why Space efficient?
  - What we mean by efficient? (information theory lower bound)
  - How Some examples

Overview

- Plan of the talk
  - Why Space efficient?
  - What we mean by efficient? (information theory lower bound)
  - How Some examples
    (a binary (or *d*-ary) vector, subset of a finite universe)

Overview

- Plan of the talk
  - Why Space efficient?
  - What we mean by efficient? (information theory lower bound)
  - How Some examples
    (a binary (or d-ary) vector, subset of a finite universe)
  - Success Story BWT and FM index

Overview

- Plan of the talk
  - Why Space efficient?
  - What we mean by efficient? (information theory lower bound)
  - How Some examples
    (a binary (or d-ary) vector, subset of a finite universe)
  - Success Story BWT and FM index
- A recent book
  - *Compact Data Structures: A Practical Approach*, Gonzalo Navarro, Cambridge UP, 2016.

Data Structures

Data Structures

- Pre-process input data so as to answer (long) series of *retrieval* or *update* operations.

Data Structures

- Pre-process input data so as to answer (long) series of *retrieval* or *update* operations.
- Want to minimize:
  1. Query/Update time.
  2. Space usage of data structure.
  3. Time of pre-processing.
  4. Space for pre-processing.

Data Structures

- Pre-process input data so as to answer (long) series of *retrieval* or *update* operations.
- Want to minimize:
  1. Query/Update time.
  2. Space usage of data structure.
  3. Time of pre-processing.
  4. Space for pre-processing.
- In this talk we will worry only about the first two, and

Data Structures

- Pre-process input data so as to answer (long) series of *retrieval* or *update* operations.
- Want to minimize:
  1. Query/Update time.
  2. Space usage of data structure.
  3. Time of pre-processing.
  4. Space for pre-processing.
- In this talk we will worry only about the first two, and our data structures are static.

Space usage of Data Structures

Answering queries on data requires an *index* **in addition to** the data. Index may be much larger than the data. E.g.:

Space usage of Data Structures

Answering queries on data requires an *index* **in addition to** the data. Index may be much larger than the data. E.g.:

- *Range Trees:* data structure for answering 2-D orthogonal range queries on n points.
  - Good worst-case performance but $\Theta(n \log n)$ words of space.

Space usage of Data Structures

Answering queries on data requires an *index* **in addition to** the data. Index may be much larger than the data. E.g.:

- *Range Trees:* data structure for answering 2-D orthogonal range queries on n points.
  - Good worst-case performance but $\Theta(n \log n)$ words of space.
- *Suffix Trees:* data structure for indexing a sequence T of n symbols from an alphabet of size σ.

Space usage of Data Structures

Answering queries on data requires an *index* **in addition to** the data. Index may be much larger than the data. E.g.:

- *Range Trees:* data structure for answering 2-D orthogonal range queries on n points.
  - Good worst-case performance but $\Theta(n \log n)$ words of space.
- *Suffix Trees:* data structure for indexing a sequence T of n symbols from an alphabet of size σ.
  - Supports very complex queries on string patterns quickly but uses $\Theta(n)$ words of space.
  - One word must have at least $\log_2 n$ bits.
  - $\Theta(n)$ words is $\Omega(n \log n)$ bits – raw sequence T is $n \log_2 \sigma$ bits.
  - A good implementation takes 10x to 30x space more than T.

Succinct/Compressed Data Structures

$$\text{Space usage} = \text{``space for data''} + \underbrace{\text{``space for index''}}_{\textit{redundancy}}.$$

- Redundancy (working space used by data structure to answer queries) should be small.

Succinct/Compressed Data Structures

$$\text{Space usage} = \text{“space for data”} + \underbrace{\text{“space for index”}}_{\textit{redundancy}}.$$

- Redundancy (working space used by data structure to answer queries) should be small. Ideally $o(\textit{inputsize})$.
- What should be the space for the data?

Why care about space?

Why care about space?

- While the cost of memory continues to go down, the growth of data is increasing at a much higher rate. (E.g. Search Engines, Genome data)

Why care about space?

- While the cost of memory continues to go down, the growth of data is increasing at a much higher rate. (E.g. Search Engines, Genome data)
- Space is important if we want to pack a lot of data into handheld devices.

Why care about space?

- While the cost of memory continues to go down, the growth of data is increasing at a much higher rate. (E.g. Search Engines, Genome data)
- Space is important if we want to pack a lot of data into handheld devices.
- Sometimes, better space usage increases the amount of data that can be stored in main memory, thereby increasing time efficiency too.

Models of Computation

- Computational model:
  - Unit-cost RAM with word size $\Theta(\log n)$ bits.
    - Operations on $O(\log n)$ bit operands (addition, subtraction, OR, multiplication, ..) in $O(1)$ time.
  - Space counted in terms of bits.

Models of Computation

- Computational model:
  - Unit-cost RAM with word size $\Theta(\log n)$ bits.
    - Operations on $O(\log n)$ bit operands (addition, subtraction, OR, multiplication, ..) in $O(1)$ time.
  - Space counted in terms of bits.
  - There are also other models like Cell-probe model with word size $\Theta(\log n)$ bits (normally used for lower bounds).

"Space for Data"

Definition (Information-theoretic Lower Bound)

If an object x is chosen from a set S then in the worst case we need $\log_2 |S|$ bits to represent x.

"Space for Data"

Definition (Information-theoretic Lower Bound)

If an object x is chosen from a set S then in the worst case we need $\log_2 |S|$ bits to represent x.

- x is a binary string of length n.
- S is the set of all binary strings of length n.
- $\log_2 |S| = \log_2 2^n = n$ bits.

"Space for Data"

Definition (Information-theoretic Lower Bound)

If an object x is chosen from a set S then in the worst case we need $\log_2 |S|$ bits to represent x.

- x is a permutation over $\{1, \ldots, n\}$.
- S is the set of all permutations over $\{1, \ldots, n\}$.
- $\log_2 |S| = \log_2 n! = n \log_2 n - n \log_2 e + o(n)$ bits.

Note that the standard way to represent a permutation takes $n \lceil \lg n \rceil$ bits.

"Space for Data"

Definition (Information-theoretic Lower Bound)

If an object x is chosen from a set S then in the worst case we need $\log_2 |S|$ bits to represent x.

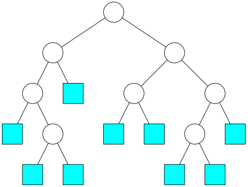
- x is a binary string of length n with m 1s.
- S is the set of all binary strings of length n with m 1s.
- $\log_2 |S| = \log_2 \binom{n}{m} = m \log_2(n/m) + O(m)$ bits.
    - E.g. if $m = O(n/\log n)$ then the lower bound is $O(m \log\log n) = o(n)$ bits.
    - if we just write down the positions of the 1's, that is $m \lceil \log_2 n \rceil$ bits

"Space for Data"

Definition (Information-theoretic Lower Bound)

If an object x is chosen from a set S then in the worst case we need $\log_2 |S|$ bits to represent x.

- x is a binary tree of n nodes.
- S is the set of all binary trees of n nodes.
- $\log_2 |S| = \log_2 \frac{1}{n+1}\binom{2n}{n} = 2n - O(\log n)$ bits



Note that the standard binary tree representation uses $\Theta(1)$ pointers per node, or $\Theta(n)$ pointers; each pointer is an address needing $\log n$ bits, so totally $\Theta(n \log n)$ bits, $\log n$ times more than necessary.

"Space for Data"

Definition (Information-theoretic Lower Bound)

If an object x is chosen from a set S then in the worst case we need $\log_2 |S|$ bits to represent x.

- x is a triangulated planar graph of n nodes.
- S is the set of all triangulated planar graphs with n nodes.
- $\log_2 |S| \sim 3.24n$ bits.

There are also bounds for general graphs, chordal graphs, bounded treewidth graphs.

Overview

Succinct Data Structures

Aim is to store using space:

$$\text{Space usage} = \text{“space for data”} + \underbrace{\text{“space for index”}}_{\textit{lower-order term}}.$$

and perform operations *directly* on it.

- For static DS, often get $O(1)$ time operations.
- Representation often tightly tied to set of operations.
- They work in practice!

Bit Vectors

Data: Sequence X of n bits, $x_1, \ldots, x_n$.

ITLB: n bits; total space $n + o(n)$ bits.

Bit Vectors

Data: Sequence X of n bits, $x_1, \ldots, x_n$.

ITLB: n bits; total space $n + o(n)$ bits.

Operations:

- $rank_1(i)$: number of 1s in $x_1, \ldots, x_i$.
- $select_1(i)$: position of ith 1.

Also $rank_0$, $select_0$. Ideally all in $O(1)$ time.

Example: $X = 01101001$, $rank_1(4) = 2$, $select_0(4) = 7$.

Bit Vectors

Data: Sequence X of n bits, $x_1, \ldots, x_n$.

ITLB: n bits; total space $n + o(n)$ bits.

Operations:

- $rank_1(i)$: number of 1s in $x_1, \ldots, x_i$.
- $select_1(i)$: position of ith 1.

Also $rank_0$, $select_0$. Ideally all in $O(1)$ time.

Example: $X = 01101001$, $rank_1(4) = 2$, $select_0(4) = 7$.

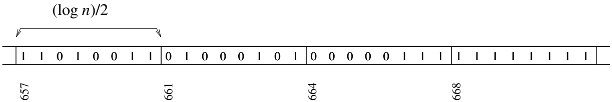
Operations introduced in [Elias, *J. ACM '75*], [Tarjan and Yao, *C. ACM '78*], [Chazelle, *SIAM J. Comput '85*], [Jacobson, *FOCS '89*].

Bit Vectors: Implementing $rank_1$

1	1	0	1	0	0	1	1	0	1	0	0	0	1	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
657	658	658	659	659	659	660	661	661	662	662	662	662	663	663	664	664	664	664	664	664	665	666	667	668	669	670	671	672	673	674	675

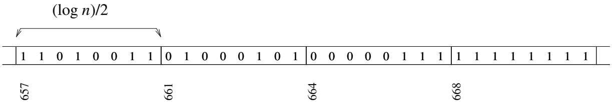
- Naive solution: store answer to all $rank_1$ queries. Space: $O(n \log n)$ bits.

Bit Vectors: Implementing $rank_1$



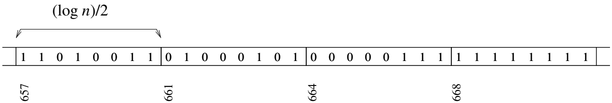
- Naive solution: store answer to all $rank_1$ queries. Space: $O(n \log n)$ bits.
- Sample: store answer only to every $(\log n)/2$-th $rank_1$ queries. Space: $O(n)$ bits.

Bit Vectors: Implementing $rank_1$

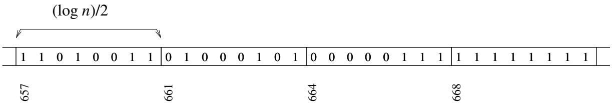


- Naive solution: store answer to all $rank_1$ queries. Space: $O(n \log n)$ bits.
- Sample: store answer only to every $(\log n)/2$-th $rank_1$ queries. Space: $O(n)$ bits.
- How to support $rank_1$ in $O(1)$ time?

Bit-Vectors: Implementing $rank_1$

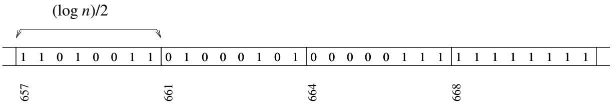


Bit-Vectors: Implementing $rank_1$



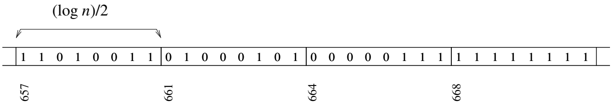
- Scanning the $(\log n)/2$ block takes $O(\log n)$ time.

Bit-Vectors: Implementing $rank_1$



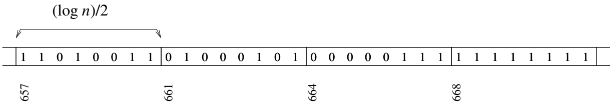
- Scanning the $(\log n)/2$ block takes $O(\log n)$ time.
- We will use what is called the "Four Russians trick".

Bit-Vectors: Implementing $rank_1$



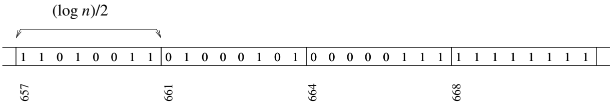
- Scanning the $(\log n)/2$ block takes $O(\log n)$ time.
- We will use what is called the “Four Russians trick”.
- Let $k = (\log n)/2$. Create a table A with $2^{k+\log_2 k} = O(\sqrt{n}\log n) = o(n)$ entries.

Bit-Vectors: Implementing $rank_1$



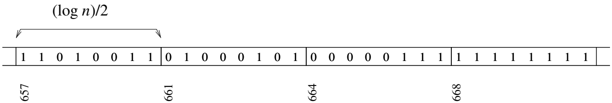
- Scanning the $(\log n)/2$ block takes $O(\log n)$ time.
- We will use what is called the "Four Russians trick".
- Let $k = (\log n)/2$. Create a table A with $2^{k+\log_2 k} = O(\sqrt{n} \log n) = o(n)$ entries.
- $A[y_1 \ldots y_{\log_2 k} x_1 \ldots x_k]$ = number of 1s in $x_1 \ldots x_{y+1}$ where $y = y_1 \ldots y_{\log_2 k}$. (The "four Russians" trick.)

Bit-Vectors: Implementing $rank_1$



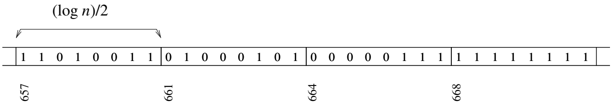
- Scanning the $(\log n)/2$ block takes $O(\log n)$ time.
- We will use what is called the “Four Russians trick”.
- Let $k = (\log n)/2$. Create a table A with $2^{k+\log_2 k} = O(\sqrt{n}\log n) = o(n)$ entries.
- $A[y_1 \dots y_{\log_2 k} x_1 \dots x_k]$ = number of 1s in $x_1 \dots x_{y+1}$ where $y = y_1 \dots y_{\log_2 k}$. (The “four Russians” trick.)
- $rank_1(x) = 657 + \underbrace{A[10111010011]}_{3}$.

Bit-Vectors: Implementing $rank_1$



- Scanning the $(\log n)/2$ block takes $O(\log n)$ time.
- We will use what is called the "Four Russians trick".
- Let $k = (\log n)/2$. Create a table A with $2^{k+\log_2 k} = O(\sqrt{n} \log n) = o(n)$ entries.
- $A[y_1 \ldots y_{\log_2 k} x_1 \ldots x_k]$ = number of 1s in $x_1 \ldots x_{y+1}$ where $y = y_1 \ldots y_{\log_2 k}$. (The "four Russians" trick.)
- $rank_1(x) = 657 + \underbrace{A[10111010011]}_{3}$.
- $O(n)$ bits, $O(1)$ time.

Bit-Vectors: Implementing $rank_1$



- Scanning the $(\log n)/2$ block takes $O(\log n)$ time.
- We will use what is called the "Four Russians trick".
- Let $k = (\log n)/2$. Create a table A with $2^{k+\log_2 k} = O(\sqrt{n}\log n) = o(n)$ entries.
- $A[y_1 \dots y_{\log_2 k} x_1 \dots x_k]$ = number of 1s in $x_1 \dots x_{y+1}$ where $y = y_1 \dots y_{\log_2 k}$. (The "four Russians" trick.)
- $rank_1(x) = 657 + \underbrace{A[10111010011]}_{3}$.
- $O(n)$ bits, $O(1)$ time.
- Many theoretical SDS: decompose + sample + table lookup.

Bit-Vectors: Implementing $rank_1$

Improve redundancy by two-level approach.

Bit-Vectors: Implementing $rank_1$

Improve redundancy by two-level approach.

- Store answer for every $\log^2 n$ positions. This takes only $O(n \log n / \log^2 n = n / \log n) = o(n)$ bits.

Bit-Vectors: Implementing $rank_1$

Improve redundancy by two-level approach.

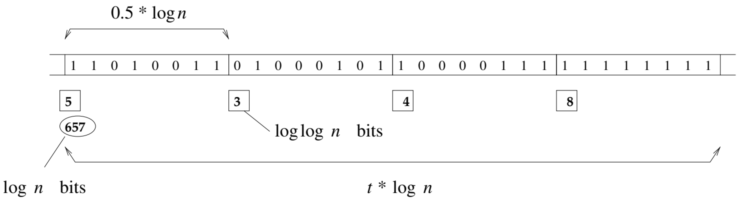
- Store answer for every $\log^2 n$ positions. This takes only $O(n \log n / \log^2 n = n / \log n) = o(n)$ bits.
- Then for every $(\log n)/2$ positions, store answer **within** the block. This takes $O(n(\log \log n) / \log n) = o(n)$ bits.

Bit-Vectors: Implementing $rank_1$

Improve redundancy by two-level approach.

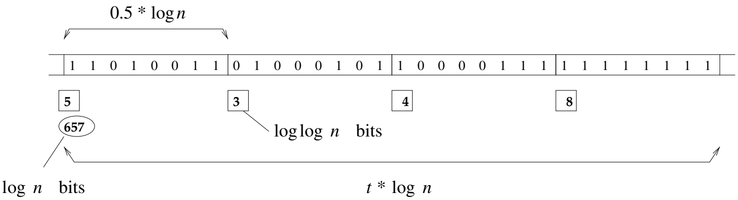
- Store answer for every $\log^2 n$ positions. This takes only $O(n \log n / \log^2 n = n / \log n) = o(n)$ bits.
- Then for every $(\log n)/2$ positions, store answer **within** the block. This takes $O(n(\log \log n) / \log n) = o(n)$ bits.
- Then store, as before, a table to find answers within $(\log n)/2$ positions.

Bit-Vectors: Implementing $rank_1$ Two-level approach



$$
\begin{aligned}
\text{Space} &= n + O\left(\frac{n}{t \lg n} \cdot \lg n + \frac{n}{\lg n} \cdot \lg \lg n\right) + O(\sqrt{n} \cdot \lg n) \\
&= n + O(n \log \log n / \log n) \text{ bits: choose } t = \Theta(\log n / \log \log n).
\end{aligned}
$$

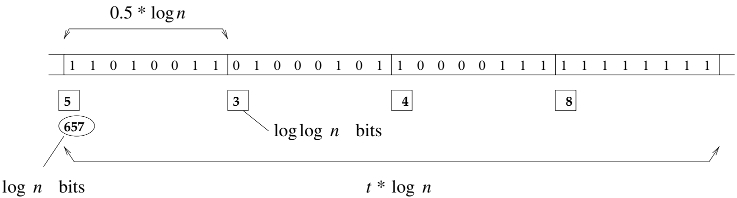
Bit-Vectors: Implementing $rank_1$ Two-level approach



$$\begin{aligned}\text{Space} &= n + O\left(\frac{n}{t \lg n} \cdot \lg n + \frac{n}{\lg n} \cdot \lg\lg n\right) + O(\sqrt{n} \cdot \lg n) \\ &= n + O(n \log\log n / \log n) \text{ bits: choose } t = \Theta(\log n / \log\log n).\end{aligned}$$

- Redundancy $O(n \lg\lg n / \lg n)$ bits, *optimal* for $O(1)$ time operations [Golynski, *TCS'07*].

Bit-Vectors: Implementing $rank_1$ Two-level approach



$$\text{Space} = n + O\left(\frac{n}{t \lg n} \cdot \lg n + \frac{n}{\lg n} \cdot \lg\lg n\right) + O(\sqrt{n} \cdot \lg n)$$

$$= n + O(n \log\log n / \log n) \text{ bits: choose } t = \Theta(\log n / \log\log n).$$

- Redundancy $O(n \lg\lg n / \lg n)$ bits, *optimal* for $O(1)$ time operations [Golynski, *TCS'07*].
- Supporting $select_1$ is similar, though a bit complicated.

Bit-Vectors: Implementing $select_1$; the idea

Bit-Vectors: Implementing $select_1$; the idea

- We will try to manage by using extra $O(n/\log\log n)$ bits.

Bit-Vectors: Implementing $select_1$; the idea

- We will try to manage by using extra $O(n/\log\log n)$ bits.
- Store answer for every $\lg n(\lg\lg n)^{th}$ 1, takes space $n/\lg lgn$ bits.

Bit-Vectors: Implementing $select_1$; the idea

- We will try to manage by using extra $O(n/\log\log n)$ bits.
- Store answer for every $\lg n(\lg\lg n)^{th}$ 1, takes space $n/\lg lgn$ bits.
- If the range r between two consecutive answers stored is of size more than $(\lg n \lg\lg n)^2$, store the positions of all the $\lg n(\lg\lg n)$ 1 in the range;

Bit-Vectors: Implementing $select_1$; the idea

- We will try to manage by using extra $O(n/\log\log n)$ bits.
- Store answer for every $\lg n(\lg\lg n)^{th}$ 1, takes space $n/\lg lgn$ bits.
- If the range r between two consecutive answers stored is of size more than $(\lg n \lg\lg n)^2$, store the positions of all the $\lg n(\lg\lg n)$ 1 in the range; takes $(\lg n)^2(\lg\lg n)$ bits, which is at most $r/\lg\lg n$.

Bit-Vectors: Implementing $select_1$; the idea

- We will try to manage by using extra $O(n/\log\log n)$ bits.
- Store answer for every $\lg n(\lg\lg n)^{th}$ 1, takes space $n/\lg lgn$ bits.
- If the range r between two consecutive answers stored is of size more than $(\lg n \lg\lg n)^2$, store the positions of all the $\lg n(\lg\lg n)$ 1 in the range; takes $(\lg n)^2(\lg\lg n)$ bits, which is at most $r/\lg\lg n$.
- Otherwise recurse.

Bit-Vectors: Implementing $select_1$; the idea

- We will try to manage by using extra $O(n/\log\log n)$ bits.
- Store answer for every $\lg n(\lg\lg n)^{th}$ 1, takes space $n/\lg lgn$ bits.
- If the range r between two consecutive answers stored is of size more than $(\lg n\lg\lg n)^2$, store the positions of all the $\lg n(\lg\lg n)$ 1 in the range; takes $(\lg n)^2(\lg\lg n)$ bits, which is at most $r/\lg\lg n$.
- Otherwise recurse. After a couple of levels, the range will be small enough $(O((\lg\lg n)^4))$ that a table look up can complete the job.

Wavelet Tree – Representing strings from a larger alphabet

Wavelet Tree – Representing strings from a larger alphabet

Data: Sequence $S[1..n]$ of symbols from an alphabet of size σ.
Operations:

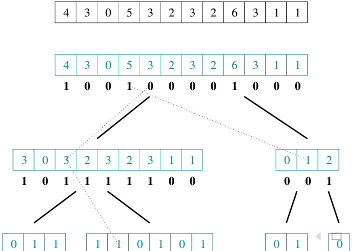
$$\left.\begin{array}{l} \mathit{rank}(c,i)\text{: number of } c\text{'s in } S[1..i]. \\ \mathit{select}(c,i)\text{: position of } i\text{-th } c. \\ \mathit{access}(i)\text{: return } S[i]. \end{array}\right\} \text{in } O(\log\sigma) \text{ time.}$$

Wavelet Tree – Representing strings from a larger alphabet

Data: Sequence $S[1..n]$ of symbols from an alphabet of size σ.
Operations:

$$\left.\begin{array}{l} \mathit{rank}(c,i)\text{: number of } c\text{'s in } S[1..i]. \\ \mathit{select}(c,i)\text{: position of } i\text{-th } c. \\ \mathit{access}(i)\text{: return } S[i]. \end{array}\right\} \text{in } O(\log\sigma) \text{ time.}$$

Store $\log_2\sigma$ BVs: $\underbrace{n\log\sigma}_{\text{raw size}} + o(n\log\sigma)$ bits [Grossi, Vitter, *SJC '05*].



A Bit vector with only m 1s

A Bit vector with only m 1s

Data: Sequence X of n bits, $x_1, \ldots, x_n$ with m 1s.

Operations:

- $select_1(i)$.

Data: Set $X = \{x_1, \ldots, x_m\} \subseteq \{1, \ldots, n\}$, $x_1 < x_2 < \ldots < x_m$.

Operations:

- $access(i)$: return x_i.

ITLB: $\log_2 \binom{n}{m} = m \log_2(n/m) + O(m)$ bits.

[Elias, *J. ACM'75*], [Grossi/Vitter, *SICOMP'06*], [Raman et al., *TALG'07*].

Elias-Fano Representation

Bucket according to most significant b bits.

Example. $b = 3$, $\lceil \log_2 n \rceil = 5$, $m = 7$.

x_1	0	1	0	0	0
x_2	0	1	0	0	1
x_3	0	1	0	1	1
x_4	0	1	1	0	1
x_5	1	0	0	0	0
x_6	1	0	0	1	0
x_7	1	0	1	1	1

Bucket	Keys
000	—
001	—
010	x_1, x_2, x_3
011	x_4
100	x_5, x_6
101	x_7
110	—
111	—

Elias-Fano

- ▷ Store only low-order bits.
- ▷ Keep sizes of all buckets.

Example

select(6)

bkt	sz	data
000	0	—
001	0	—
010	3	$\underbrace{00}_{x_1}, \underbrace{01}_{x_2}, \underbrace{11}_{x_3},$
011	1	$\underbrace{01}_{x_4}$
100	2	$\underbrace{00}_{x_5}, \underbrace{10}_{x_6}$
101	1	$\underbrace{11}_{x_7}$
110	0	—
111	0	—

Elias-Fano

Elias-Fano

- Choose $b = \lfloor \log_2 m \rfloor$ bits. In bucket: $\lceil \log_2 n \rceil - \lfloor \log_2 m \rfloor$-bit keys.

Elias-Fano

- Choose $b = \lfloor \log_2 m \rfloor$ bits. In bucket: $\lceil \log_2 n \rceil - \lfloor \log_2 m \rfloor$-bit keys.
- $m \log_2 n - m \log_2 m + O(m) = m \log_2 (n/m) + O(m)$ bits for lower part.

Elias-Fano

- Choose $b = \lfloor \log_2 m \rfloor$ bits. In bucket: $\lceil \log_2 n \rceil - \lfloor \log_2 m \rfloor$-bit keys.
- $m \log_2 n - m \log_2 m + O(m) = m \log_2(n/m) + O(m)$ bits for lower part.

Encoding Bucket Sizes

Bucket no:	000	001	010	011	100	101	110	111
Bucket size:	0	0	3	1	2	1	0	0

- Use a *unary* encoding: $0, 0, 3, 1, 2, 1, 0, 0 \rightarrow$ 110001010010111.

Elias-Fano

- Choose $b = \lfloor \log_2 m \rfloor$ bits. In bucket: $\lceil \log_2 n \rceil - \lfloor \log_2 m \rfloor$-bit keys.
- $m \log_2 n - m \log_2 m + O(m) = m \log_2(n/m) + O(m)$ bits for lower part.

Encoding Bucket Sizes

Bucket no:	000	001	010	011	100	101	110	111
Bucket size:	0	0	3	1	2	1	0	0

- Use a *unary* encoding: $0, 0, 3, 1, 2, 1, 0, 0 \rightarrow$ 110001010010111.
- z buckets, total size $m \Rightarrow m + z = O(m)$ bits ($z = 2^{\lfloor \log_2 m \rfloor}$).
  - Overall space of E-F bit-vector is $m \log(n/m) + O(m)$ bits.
- In which bucket is the 6th key? ▷ "$rank_1$ of 6th 0".
  - $select_1$ in $O(1)$ time.

Elias-Fano

- Choose $b = \lfloor \log_2 m \rfloor$ bits. In bucket: $\lceil \log_2 n \rceil - \lfloor \log_2 m \rfloor$-bit keys.
- $m \log_2 n - m \log_2 m + O(m) = m \log_2(n/m) + O(m)$ bits for lower part.

Encoding Bucket Sizes

Bucket no:	000	001	010	011	100	101	110	111
Bucket size:	0	0	3	1	2	1	0	0

- Use a *unary* encoding: $0, 0, 3, 1, 2, 1, 0, 0 \rightarrow 110001010010111$.
- z buckets, total size $m \Rightarrow m + z = O(m)$ bits ($z = 2^{\lfloor \log_2 m \rfloor}$).
  - Overall space of E-F bit-vector is $m \log(n/m) + O(m)$ bits.
- In which bucket is the 6th key? $\triangleright$ "$rank_1$ of 6th 0".
  - $select_1$ in $O(1)$ time.
  - Redundancy can be made $o(m)$ and membership and *Rankone* can also be supported (RRR01)

Tree Representations

Tree Representations

Data: n-node binary tree.

Tree Representations

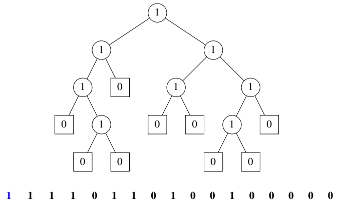
Data: n-node binary tree.

Operations: Navigation (left child, right child, parent).

Tree Representations

Data: n-node binary tree.
Operations: Navigation (left child, right child, parent).

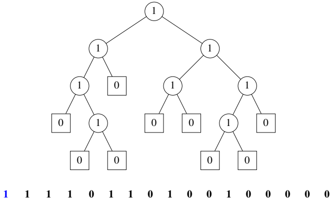
- Visit nodes in level-order and output 1 if internal node and 0 if external ($2n + 1$ bits) [Jacobson, *FOCS '89*]. Store sequence of bits as bit vector.



Tree Representations

Data: n-node binary tree.
Operations: Navigation (left child, right child, parent).

- Visit nodes in level-order and output 1 if internal node and 0 if external ($2n + 1$ bits) [Jacobson, *FOCS '89*]. Store sequence of bits as bit vector.



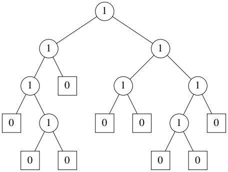
- Number internal nodes by position of 1 in bit-string

Tree Representations

Data: n-node binary tree.

Operations: Navigation (left child, right child, parent).

- Visit nodes in level-order and output 1 if internal node and 0 if external ($2n+1$ bits) [Jacobson, *FOCS '89*]. Store sequence of bits as bit vector.



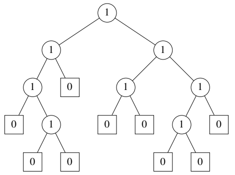
1 1 1 1 0 1 1 0 1 0 0 1 0 0 0 0 0

- Number internal nodes by position of 1 in bit-string
- Left child $= 2 * rank_1(i)$.

Tree Representations

Data: n-node binary tree.
Operations: Navigation (left child, right child, parent).

- Visit nodes in level-order and output 1 if internal node and 0 if external ($2n+1$ bits) [Jacobson, *FOCS '89*]. Store sequence of bits as bit vector.



- Number internal nodes by position of 1 in bit-string
- Left child $= 2 * rank_1(i)$. E.g. Left child of node 7 $= 7 * 2 = 14$. Right child $= 2 * rank_1(i) + 1$. parent $= select_1(\lfloor i/2 \rfloor)$.

Tree Representations

Tree Representations

- "Optimal" representations of many kinds of trees e.g. ordinal trees (rooted arbitrary degree (un-)labelled trees, e.g. XML documents), tries.
- Wide range of $O(1)$-time operations, e.g.:
  - ordinal trees in $2n + o(n)$ bits [Navarro, Sadakane, *TALG'12*].

Tree Representations

Pattern Matching – Compressed Text Indexing

Pattern Matching – Compressed Text Indexing

Data: Sequence T ("text") of m symbols from alphabet of size σ.
ITLB: $n \log_2 \sigma$ bits.
Operation: Given pattern P, determine if P occurs (exactly) in T (and report the number of occurrences, starting positions etc).

Pattern Matching – Compressed Text Indexing

Data: Sequence T ("text") of m symbols from alphabet of size σ.
ITLB: $n \log_2 \sigma$ bits.
Operation: Given pattern P, determine if P occurs (exactly) in T (and report the number of occurrences, starting positions etc).

- For a human genome sequence, m is about 3 billion ($3x10^9$) characters, and $\sigma = 4$.

Pattern Matching – Compressed Text Indexing

Data: Sequence T ("text") of m symbols from alphabet of size σ.
ITLB: $n \log_2 \sigma$ bits.
Operation: Given pattern P, determine if P occurs (exactly) in T (and report the number of occurrences, starting positions etc).

- For a human genome sequence, m is about 3 billion ($3x10^9$) characters, and $\sigma = 4$.
- Standard data structure is *suffix tree*, which answers this query in $O(|P|)$ time but takes $O(n \log n)$ bits of space.
- In practice, a ST is about 10-30 times larger than the text.

Pattern Matching – Compressed Text Indexing

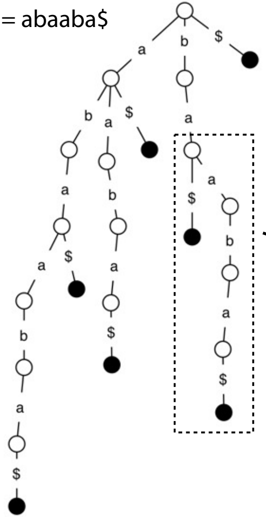
Data: Sequence T ("text") of m symbols from alphabet of size σ.
ITLB: $n \log_2 \sigma$ bits.
Operation: Given pattern P, determine if P occurs (exactly) in T (and report the number of occurrences, starting positions etc).

- For a human genome sequence, m is about 3 billion ($3x10^9$) characters, and $\sigma = 4$.
- Standard data structure is *suffix tree*, which answers this query in $O(|P|)$ time but takes $O(n \log n)$ bits of space.
- In practice, a ST is about 10-30 times larger than the text.
- A number of SDS have been developed: we'll focus on the FM-Index [Ferragina, Manzini, *JACM '05*].

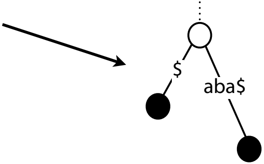
Previous Popular Solution – Suffix Trees

Suffix trie: making it smaller

T = abaaba$



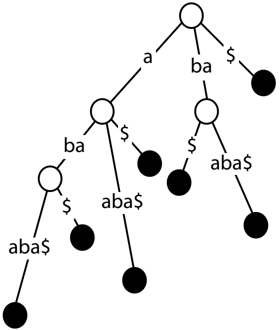
Idea 1: Coalesce non-branching paths into a *single edge* with a *string* label



Reduces # nodes, edges,
guarantees internal nodes have >1 child

Suffix tree

T = abaaba$



With respect to m:

How many leaves? ***m***

How many non-leaf nodes? ***≤ m - 1***

≤ 2m -1 nodes total, or $O(m)$ nodes

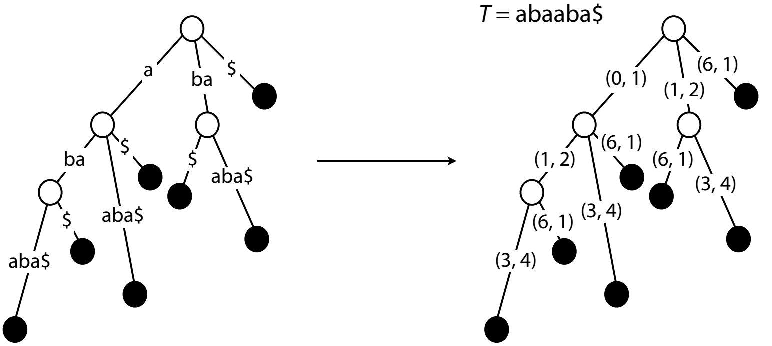
Is the total size $O(m)$ now?

No: total length of edge labels is quadratic in m

Suffix tree

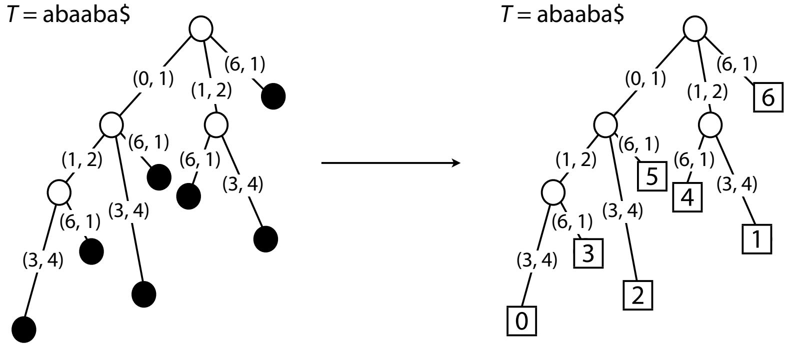
T = abaaba\$

Idea 2: Store T itself in addition to the tree. Convert tree's edge labels to (offset, length) pairs with respect to T.



Space required for suffix tree is now $O(m)$

Suffix tree: leaves hold offsets



Previous Popular Solution – Suffix Trees

Previous Popular Solution – Suffix Trees

- A (compressed) trie containing all the suffixes of T. The tree contains $m+1$ leaves and at most m other nodes.

Previous Popular Solution – Suffix Trees

- A (compressed) trie containing all the suffixes of T. The tree contains $m + 1$ leaves and at most m other nodes.
- Each leaf is labelled with the starting position of the suffix ending at that leaf.

Previous Popular Solution – Suffix Trees

- A (compressed) trie containing all the suffixes of T. The tree contains $m + 1$ leaves and at most m other nodes.
- Each leaf is labelled with the starting position of the suffix ending at that leaf.
- Each edge has a string, that can be represented by the starting and ending position of the substring in the text.

Previous Popular Solution – Suffix Trees

- A (compressed) trie containing all the suffixes of T. The tree contains $m+1$ leaves and at most m other nodes.
- Each leaf is labelled with the starting position of the suffix ending at that leaf.
- Each edge has a string, that can be represented by the starting and ending position of the substring in the text.
- Overall, naive implementation takes about $4n$ words or $4n \lg n$ bits.

Previous Popular Solution – Suffix Trees

- A (compressed) trie containing all the suffixes of T. The tree contains $m+1$ leaves and at most m other nodes.
- Each leaf is labelled with the starting position of the suffix ending at that leaf.
- Each edge has a string, that can be represented by the starting and ending position of the substring in the text.
- Overall, naive implementation takes about $4n$ words or $4n \lg n$ bits.
- Progress in succinct data structures has brought the space down to $m \lg m + O(m)$ bits (in addition to the text).

Previous Popular Solution – Suffix Trees

- A (compressed) trie containing all the suffixes of T. The tree contains $m+1$ leaves and at most m other nodes.
- Each leaf is labelled with the starting position of the suffix ending at that leaf.
- Each edge has a string, that can be represented by the starting and ending position of the substring in the text.
- Overall, naive implementation takes about $4n$ words or $4n \lg n$ bits.
- Progress in succinct data structures has brought the space down to $m \lg m + O(m)$ bits (in addition to the text).
- P exists in T if and only if P is a prefix of a suffix of T. So, follow from the root matching P. If success, the leaves in the entire subtree gives the list of occurrences.

Previous Popular Solution – Suffix Trees

- A (compressed) trie containing all the suffixes of T. The tree contains $m+1$ leaves and at most m other nodes.
- Each leaf is labelled with the starting position of the suffix ending at that leaf.
- Each edge has a string, that can be represented by the starting and ending position of the substring in the text.
- Overall, naive implementation takes about $4n$ words or $4n \lg n$ bits.
- Progress in succinct data structures has brought the space down to $m \lg m + O(m)$ bits (in addition to the text).
- P exists in T if and only if P is a prefix of a suffix of T. So, follow from the root matching P. If success, the leaves in the entire subtree gives the list of occurrences.
- $O(n + occ)$ to find all occurrences

Previous popular solution - Suffix Arrays

Suffix array

$T\$$ = abaaba$

As with suffix tree, T is part of index

SA(T) =
(SA = "Suffix Array")

SA(T)	Suffix
6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

$m + 1$ integers

Suffix array of T is an array of integers in $[0, m]$ specifying the lexicographic order of $T\$$'s suffixes

Suffix array: querying

Is *P* a substring of *T*?

1. For *P* to be a substring, it must be a prefix of ≥1 of *T*'s suffixes
2. Suffixes sharing a prefix are consecutive in the suffix array

Use binary search

6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

Suffix array: querying

Is *P* a substring of *T*?

Do binary search, check whether *P* is a prefix of the suffix there

How many times does *P* occur in *T*?

Two binary searches yield the range of suffixes with *P* as prefix; size of range equals # times *P* occurs in *T*

Worst-case time bound?

$O(\log_2 m)$ bisections, $O(n)$ comparisons per bisection, so $O(n \log m)$

6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

Previous popular solution - Suffix Arrays

Previous popular solution - Suffix Arrays

- A permutation of $\{1, 2, \ldots m\}$. $S[i]$ is the starting position of the i-th suffix in the lexicographic order.

Previous popular solution - Suffix Arrays

- A permutation of $\{1, 2, \ldots m\}$. $S[i]$ is the starting position of the i-th suffix in the lexicographic order.
- Takes $m \lg m$ bits.

Previous popular solution - Suffix Arrays

- A permutation of $\{1, 2, \dots m\}$. $S[i]$ is the starting position of the i-th suffix in the lexicographic order.
- Takes $m \lg m$ bits. Naive binary search takes $O(n \lg m)$ time.

Previous popular solution - Suffix Arrays

- A permutation of $\{1, 2, \ldots m\}$. $S[i]$ is the starting position of the i-th suffix in the lexicographic order.
- Takes $m \lg m$ bits. Naive binary search takes $O(n \lg m)$ time.
- With what is called an LCP array taking another $m \lg m$ bits, the search time can be brought down to $O(n + \lg m)$ bits.

The FM-Index

The FM-Index

Based on the Burrows-Wheeler transform of the text T.

Example: T = missisippi

F										L
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	i	s	s	i	p	p	i	m
m	i	s	s	i	s	s	i	p	p	i
p	i	m	i	s	s	i	s	s	i	p
p	p	i	m	i	s	s	i	s	s	i
s	i	p	p	i	m	i	s	s	i	s
s	i	s	s	i	p	p	i	m	i	s
s	s	i	p	p	i	m	i	s	s	i
s	s	i	s	s	i	p	p	i	m	i

BWT(T) = pssmipissii

Burrows-Wheeler Transform

Text transform that is useful for compression & search.

banana		
banana$		$banana
anana$b		a$banan
nana$ba	sort $\longrightarrow$	ana$ban
ana$ban		anana$b
na$bana		banana$
a$banan		nana$ba
$banana		na$bana

BWT(banana) =
annb$aa

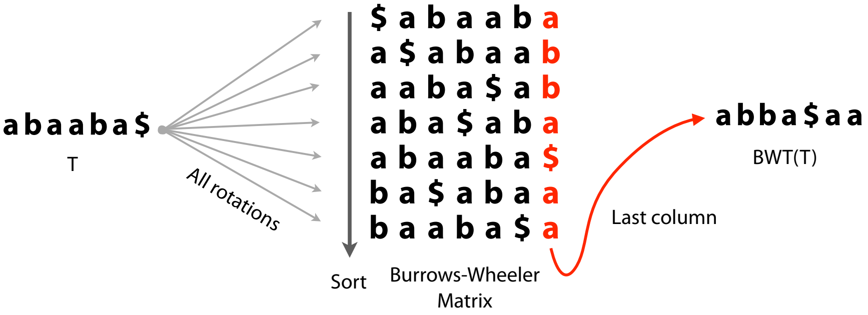
Tends to put runs of the same character together.

Makes compression work well.

"bzip" is based on this.

Burrows-Wheeler Transform

Reversible permutation of the characters of a string, used originally for compression



How is it useful for compression? How is it reversible? How is it an index?

Burrows M, Wheeler DJ: A block sorting lossless data compression algorithm. *Digital Equipment Corporation, Palo Alto, CA* 1994, Technical Report 124; 1994

Burrows-Wheeler Transform

BWM bears a resemblance to the suffix array

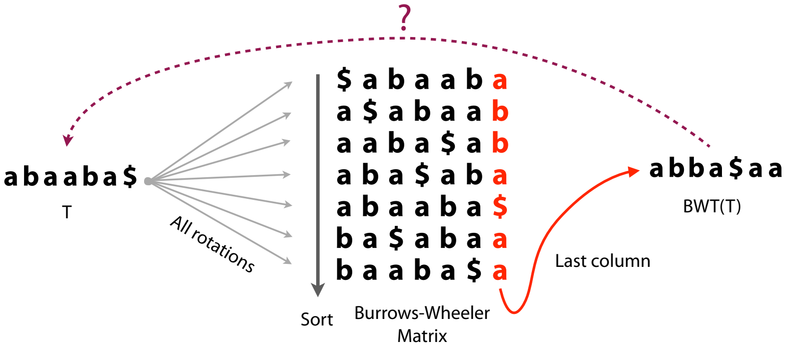
BWM(T)
$ a b a a b a
a $ a b a a b
a a b a $ a b
a b a $ a b a
a b a a b a $
b a $ a b a a
b a a b a $ a

SA(T)	
6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

Sort order is the same whether rows are rotations or suffixes

Burrows-Wheeler Transform

How to reverse the BWT?



BWM has a key property called the *LF Mapping*...

Burrows-Wheeler Transform: T-ranking

Give each character in *T* a rank, equal to # times the character occurred previously in *T*. Call this the *T-ranking*.

$$\mathbf{a_0\ b_0\ a_1\ a_2\ b_1\ a_3\ \$}$$

Now let's re-write the BWM including ranks...

Burrows-Wheeler Transform

BWM with T-ranking:

$$
\begin{array}{ccccccc}
F & & & & & & L \\
\$ & a_0 & b_0 & a_1 & a_2 & b_1 & a_3 \\
a_3 & \$ & a_0 & b_0 & a_1 & a_2 & b_1 \\
a_1 & a_2 & b_1 & a_3 & \$ & a_0 & b_0 \\
a_2 & b_1 & a_3 & \$ & a_0 & b_0 & a_1 \\
a_0 & b_0 & a_1 & a_2 & b_1 & a_3 & \$ \\
b_1 & a_3 & \$ & a_0 & b_0 & a_1 & a_2 \\
b_0 & a_1 & a_2 & b_1 & a_3 & \$ & a_0
\end{array}
$$

Look at first and last columns, called *F* and *L*

And look at just the **a**s

as occur in the same order in *F* and *L*. As we look down columns, in both cases we see: a_3, a_1, a_2, a_0

Burrows-Wheeler Transform

BWM with T-ranking:

	F						L
	$	a_0	b_0	a_1	a_2	b_1	a_3
	a_3	$	a_0	b_0	a_1	a_2	$\mathbf{b_1}$
	a_1	a_2	b_1	a_3	$	a_0	$\mathbf{b_0}$
	a_2	b_1	a_3	$	a_0	b_0	a_1
	a_0	b_0	a_1	a_2	b_1	a_3	$
	$\mathbf{b_1}$	a_3	$	a_0	b_0	a_1	a_2
	$\mathbf{b_0}$	a_1	a_2	b_1	a_3	$	a_0

Same with **b**s: $\mathbf{b_1}$, $\mathbf{b_0}$

Burrows-Wheeler Transform: LF Mapping

BWM with T-ranking:

F						L
$\$$	a_0	b_0	a_1	a_2	b_1	a_3
a_3	$\$$	a_0	b_0	a_1	a_2	b_1
a_1	a_2	b_1	a_3	$\$$	a_0	b_0
a_2	b_1	a_3	$\$$	a_0	b_0	a_1
a_0	b_0	a_1	a_2	b_1	a_3	$\$$
b_1	a_3	$\$$	a_0	b_0	a_1	a_2
b_0	a_1	a_2	b_1	a_3	$\$$	a_0

LF Mapping: The i^{th} occurrence of a character c in L and the i^{th} occurrence of c in F correspond to the *same* occurrence in T

However we rank occurrences of c, ranks appear in the same order in F and L

Burrows-Wheeler Transform: LF Mapping

Why does the LF Mapping hold?

Why are these **a**s in this order relative to each other?

$\$$	a	b	a	a	b	a_3
a_3	$\$$	a	b	a	a	b_1
a_1	a	b	a	$\$$	a	b_0
a_2	b	a	$\$$	a	b	a_1
a_0	b	a	a	b	a	$\$$
b_1	a	$\$$	a	b	a	a_2
b_0	a	a	b	a	$\$$	a_0

They're sorted by right-context

$\$$	a	b	a	a	b	a_3
a_3	$\$$	a	b	a	a	b_1
a_1	a	b	a	$\$$	a	b_0
a_2	b	a	$\$$	a	b	a_1
a_0	b	a	a	b	a	$\$$
b_1	a	$\$$	a	b	a	a_2
b_0	a	a	b	a	$\$$	a_0

Why are these **a**s in this order relative to each other?

They're sorted by right-context

Occurrences of *c* in *F* are sorted by right-context. Same for *L*!

Whatever ranking we give to characters in *T*, rank orders in *F* and *L* will match

Burrows-Wheeler Transform: LF Mapping

BWM with T-ranking:

F						*L*
$	a_0	b_0	a_1	a_2	b_1	a_3
a_3	$	a_0	b_0	a_1	a_2	b_1
a_1	a_2	b_1	a_3	$	a_0	b_0
a_2	b_1	a_3	$	a_0	b_0	a_1
a_0	b_0	a_1	a_2	b_1	a_3	$
b_1	a_3	$	a_0	b_0	a_1	a_2
b_0	a_1	a_2	b_1	a_3	$	a_0

We'd like a different ranking so that for a given character, ranks are in ascending order as we look down the F / L columns...

Burrows-Wheeler Transform: LF Mapping

BWM with B-ranking:

F						L
$	a_3	b_1	a_1	a_2	b_0	a_0
a_0	$	a_3	b_1	a_1	a_2	b_0
a_1	a_2	b_0	a_3	$	a_3	b_1
a_2	b_0	a_0	$	a_3	b_1	a_1
a_3	b_1	a_1	a_2	b_0	a_0	$
b_0	a_0	$	a_3	b_1	a_1	a_2
b_1	a_1	a_2	b_0	a_0	$	a_3

Ascending rank

F now has very simple structure: a **$**, a block of **a**s *with ascending ranks*, a block of **b**s *with ascending ranks*

Burrows-Wheeler Transform

	F	*L*
	$	$\mathbf{a}_0$
	$\mathbf{a}_0$	$\mathbf{b}_0$
	$\mathbf{a}_1$	$\mathbf{b}_1$ ⟵ Which BWM row *begins* with $\mathbf{b}_1$?
	$\mathbf{a}_2$	$\mathbf{a}_1$
	$\mathbf{a}_3$	**$**
	$\mathbf{b}_0$	$\mathbf{a}_2$
row 6 ⟶	$\mathbf{b}_1$	$\mathbf{a}_3$

Skip row starting with **$** (1 row)

Skip rows starting with **a** (4 rows)

Skip row starting with $\mathbf{b}_0$ (1 row)

Answer: row 6

Burrows-Wheeler Transform

Say *T* has 300 **A**s, 400 **C**s, 250 **G**s and 700 **T**s and **$** < **A** < **C** < **G** < **T**

Which BWM row (0-based) begins with $\mathbf{G}_{100}$? (Ranks are B-ranks.)

Skip row starting with **$** (1 row)
Skip rows starting with **A** (300 rows)
Skip rows starting with **C** (400 rows)
Skip first 100 rows starting with **G** (100 rows)

Answer: row 1 + 300 + 400 + 100 = **row 801**

Burrows-Wheeler Transform: reversing

Reverse BWT(T) starting at right-hand-side of *T* and moving left

Start in first row. *F* must have **\$**. *L* contains character just prior to **\$**: $\mathbf{a}_0$

$\mathbf{a}_0$: LF Mapping says this is same occurrence of **a** as first **a** in *F*. Jump to row *beginning* with $\mathbf{a}_0$. *L* contains character just prior to $\mathbf{a}_0$: $\mathbf{b}_0$.

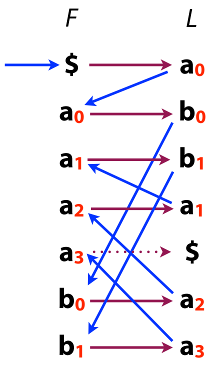
Repeat for $\mathbf{b}_0$, get $\mathbf{a}_2$

Repeat for $\mathbf{a}_2$, get $\mathbf{a}_1$

Repeat for $\mathbf{a}_1$, get $\mathbf{b}_1$

Repeat for $\mathbf{b}_1$, get $\mathbf{a}_3$

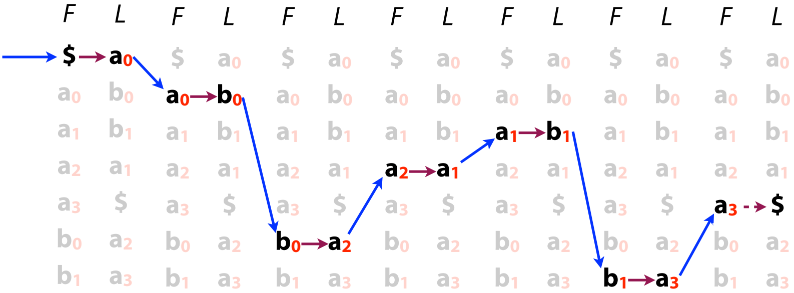
Repeat for $\mathbf{a}_3$, get **\$**, done



Reverse of chars we visited = $\mathbf{a}_3\ \mathbf{b}_1\ \mathbf{a}_1\ \mathbf{a}_2\ \mathbf{b}_0\ \mathbf{a}_0\ \$ = T$

Burrows-Wheeler Transform: reversing

Another way to visualize reversing BWT(T):



T: $a_3\ b_1\ a_1\ a_2\ b_0\ a_0\ \$$

Burrows-Wheeler Transform

We've seen how BWT is useful for compression:

- Sorts characters by right-context, making a more compressible string

And how it's reversible:

- Repeated applications of LF Mapping, recreating *T* from right to left

How is it used as an index?

FM Index

FM Index: an index combining the BWT *with a few small auxilliary data structures*

"FM" supposedly stands for "Full-text Minute-space."
(But inventors are named Ferragina and Manzini)

Core of index consists of *F* and *L* from BWM:

F can be represented very simply
(1 integer per alphabet character)

And *L* is compressible

Potentially very space-economical!

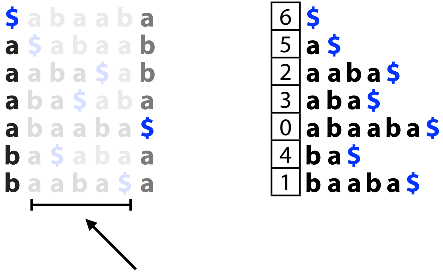
F						*L*
$	a	b	a	a	b	**a**
a	$	a	b	a	a	**b**
a	a	b	a	$	a	**b**
a	b	a	$	a	b	**a**
a	b	a	a	b	a	**$**
b	a	$	a	b	a	**a**
b	a	a	b	a	$	**a**

Not stored in index

Paolo Ferragina, and Giovanni Manzini. "Opportunistic data structures with applications." *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. IEEE, 2000.

FM Index: querying

Though BWM is related to suffix array, we can't query it the same way



We don't have these columns; binary search isn't possible

FM Index: querying

Look for range of rows of BWM(T) with *P* as prefix

Do this for *P*'s shortest suffix, then extend to successively longer suffixes until range becomes empty or we've exhausted *P*

P = **aba**

Easy to find all the rows beginning with **a**, thanks to *F*'s simple structure

F						*L*
$	a	b	a	a	b	a_3
a_0	$	a	b	a	a	b_1
a_1	a	b	a	$	a	b_0
a_2	b	a	$	a	b	a_1
a_3	b	a	a	b	a	**$**
b_0	a	$	a	b	a	a_2
b_1	a	a	b	a	$	a_0

FM Index: querying

We have rows beginning with **a**, now we seek rows beginning with **ba**

$P =$ **aba**

F						L
\$	a	b	a	a	b	a_0
a_0	\$	a	b	a	a	b_0
a_1	a	b	a	\$	a	b_1
a_2	b	a	\$	a	b	a_1
a_3	b	a	a	b	a	\$
b_0	a	\$	a	b	a	a_2
b_1	a	a	b	a	\$	a_3

← Look at those rows in *L*.

$\mathbf{b}_0$, $\mathbf{b}_1$ are **b**s occuring just to left.

Use LF Mapping. Let new range delimit those **b**s →

$P =$ **aba**

F						L
\$	a	b	a	a	b	a_0
a_0	\$	a	b	a	a	b_0
a_1	a	b	a	\$	a	b_1
a_2	b	a	\$	a	b	a_1
a_3	b	a	a	b	a	\$
b_0	a	\$	a	b	a	a_2
b_1	a	a	b	a	\$	a_3

Now we have the rows with prefix **ba**

FM Index: querying

We have rows beginning with **ba**, now we seek rows beginning with **aba**

P = **aba**

F						L
\$	a	b	a	a	b	a_0
a_0	\$	a	b	a	a	b_0
a_1	a	b	a	\$	a	b_1
a_2	b	a	\$	a	b	a_1
a_3	b	a	a	b	a	\$
b_0	a	\$	a	b	a	a_2
b_1	a	a	b	a	\$	a_3

a_2, a_3 occur just to left.

Use LF Mapping

P = **aba**

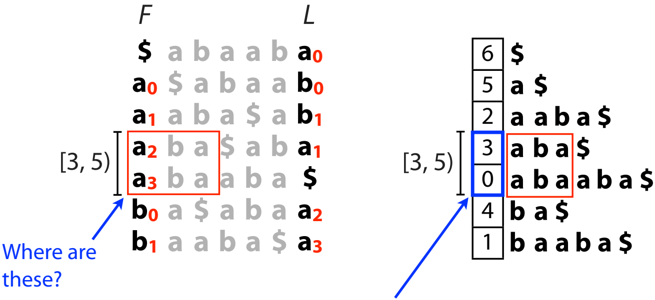
F						L
\$	a	b	a	a	b	a_0
a_0	\$	a	b	a	a	b_0
a_1	a	b	a	\$	a	b_1
a_2	b	a	\$	a	b	a_1
a_3	b	a	a	b	a	\$
b_0	a	\$	a	b	a	a_2
b_1	a	a	b	a	\$	a_3

Now we have the rows with prefix **aba**

FM Index: querying

P = **aba**

Now we have the same range, [3, 5), we would have got from querying suffix array



Unlike suffix array, we don't immediately know *where* the matches are in T...

FM Index: querying

When *P* does not occur in *T*, we will eventually fail to find the next character in *L*:

$P =$ **bba**

F						*L*
$	a	b	a	a	b	a_0
a_0	$	a	b	a	a	b_0
a_1	a	b	a	$	a	b_1
a_2	b	a	$	a	b	a_1
a_3	b	a	a	b	a	$
b_0	a	$	a	b	a	a_2
b_1	a	a	b	a	$	a_3

Rows with **ba** prefix

No **b**s!

FM Index: querying

If we *scan* characters in the last column, that can be very slow, $O(m)$

$P =$ **ab****a**

F						L
\$	a	b	a	a	b	$\mathbf{a}_3$
$\mathbf{a}_0$	\$	a	b	a	a	$\mathbf{b}_1$
$\mathbf{a}_1$	a	b	a	\$	a	$\mathbf{b}_0$
$\mathbf{a}_2$	b	a	\$	a	b	$\mathbf{a}_1$
$\mathbf{a}_3$	b	a	a	b	a	**\$**
$\mathbf{b}_0$	a	\$	a	b	a	$\mathbf{a}_2$
$\mathbf{b}_1$	a	a	b	a	\$	$\mathbf{a}_0$

Scan, looking for **b**s

FM Index: lingering issues

(1) Scanning for preceding character is slow

\$	a	b	a	a	b	a_0
a_0	\$	a	b	a	a	b_0
a_1	a	b	a	\$	a	b_1
a_2	b	a	\$	a	b	a_1
a_3	b	a	a	b	a	\$
b_0	a	\$	a	b	a	a_2
b_1	a	a	b	a	\$	a_3

$O(m)$ scan

(2) Storing ranks takes too much space

m integers

```
def reverseBwt(bw):
    """ Make T from BWT(T) """
    ranks, tots = rankBwt(bw)
    first = firstCol(tots)
    rowi = 0
    t = "$"
    while bw[rowi] != '$':
        c = bw[rowi]
        t = c + t
        rowi = first[c][0] + ranks[rowi]
    return t
```

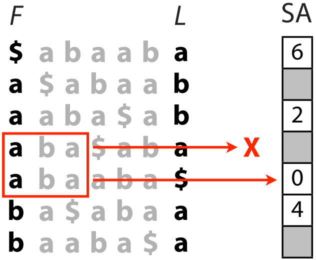
(3) Need way to find where matches occur in *T*:

\$	a	b	a	a	b	a_0
a_0	\$	a	b	a	a	b_0
a_1	a	b	a	\$	a	b_1
a_2	b	a	\$	a	b	a_1
a_3	b	a	a	b	a	\$
b_0	a	\$	a	b	a	a_2
b_1	a	a	b	a	\$	a_3

Where?

FM Index: resolving offsets

Idea: store some, but not all, entries of the suffix array



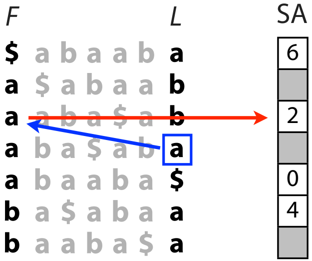
Lookup for row 4 succeeds - we kept that entry of SA

Lookup for row 3 fails - we discarded that entry of SA

FM Index: resolving offsets

But LF Mapping tells us that the **a** at the end of row 3 corresponds to...

...the **a** at the begining of row 2



And row 2 has a suffix array value = 2

So row 3 has suffix array value = 3 = 2 (row 2's SA val) + 1 (# steps to row 2)

If saved SA values are O(1) positions apart in *T*, resolving offset is O(1) time

FM Index: problems solved

Solved! At the expense of adding some SA values ($O(m)$ integers) to index

Call this the "SA sample"

(3) Need a way to find where these occurrences are in *T*:

\$	a	b	a	a	b	$\mathbf{a}_0$
$\mathbf{a}_0$	\$	a	b	a	a	$\mathbf{b}_0$
$\mathbf{a}_1$	a	b	a	\$	a	$\mathbf{b}_1$
$\mathbf{a}_2$	b	a	\$	a	b	$\mathbf{a}_1$
$\mathbf{a}_3$	b	a	a	b	a	**\$**
$\mathbf{b}_0$	a	\$	a	b	a	$\mathbf{a}_2$
$\mathbf{b}_1$	a	a	b	a	\$	$\mathbf{a}_3$

With SA sample we can do this in *O*(1) time per occurrence

To Summarize (FM index)

To Summarize (FM index)

- Existence of P in T, and
- the number of occurrences (occ) of P in T

To Summarize (FM index)

- Existence of P in T, and
- the number of occurrences (occ) of P in T

can be determined in $O(n)$ time using

To Summarize (FM index)

- Existence of P in T, and
- the number of occurrences (occ) of P in T

can be determined in $O(n)$ time using

- $m \lg \sigma$ bits, for BWT (last column)
- $o(m \lg \sigma)$ bits for rank
- $\sigma \lg m$ bits for count of each character (first column)

To Summarize (FM index)

- Existence of P in T, and
- the number of occurrences (occ) of P in T

can be determined in $O(n)$ time using

- $m \lg \sigma$ bits, for BWT (last column)
- $o(m \lg \sigma)$ bits for rank
- $\sigma \lg m$ bits for count of each character (first column)

and the position of all occurrences of P in T can be determined in

To Summarize (FM index)

- Existence of P in T, and
- the number of occurrences (occ) of P in T

can be determined in $O(n)$ time using

- $m \lg \sigma$ bits, for BWT (last column)
- $o(m \lg \sigma)$ bits for rank
- $\sigma \lg m$ bits for count of each character (first column)

and the position of all occurrences of P in T can be determined in

- additional $O(k\ occ)$ time, using
- an additional $(m \lg m)/k$ bits of space (using a sampled suffix array)

To Summarize (FM index)

- Existence of P in T, and
- the number of occurrences (occ) of P in T

can be determined in $O(n)$ time using

- $m \lg \sigma$ bits, for BWT (last column)
- $o(m \lg \sigma)$ bits for rank
- $\sigma \lg m$ bits for count of each character (first column)

and the position of all occurrences of P in T can be determined in

- additional $O(k\ occ)$ time, using
- an additional $(m \lg m)/k$ bits of space (using a sampled suffix array)
- For example, $O(occ \lg m)$ time using additional $O(m)$ bits of space.

Contrasting with Suffix Arrays and Suffix Trees

FM Index	$O(m \lg \sigma)$ bits $1.5GB$ for human genome	$O(n)$ time for finding existence and occ $O(n + occ \lg m)$ for finding all occurrences
Suffix Array	$2m \lg m$ bits + text about $12GB$ for human genome	$O(n + \lg m)$ time for all operations
Suffix Tree	$3m \lg m$ bits + text about $47GB$ in MUMmer for human genome; with optimization $(m \lg m + O(m)$ bits)	$O(n)$ time for boolean query $O(n + occ)$ for finding all occurrences useful for many other operations

Libraries

- A number of good implementations of succinct data structures in C++ are available.
- Different platforms, coding styles:
  - `sdsl-lite` (Gog, Petri et al. U. Melbourne).
  - `succinct` (Grossi and Ottaviano, U. Pisa).
  - `Sux4J` (Vigna, U. Milan, Java).
  - `LIBCDS` (Claude and Navarro, Akori and U. Chile).
- All open-source and available as `Git` repositories.

Conclusions

- SDS are a relatively mature field in terms of breadth of problems considered.

Conclusions

- SDS are a relatively mature field in terms of breadth of problems considered.
- Quite practical; FM index has been implemented in BIO software (Bowtie).

Conclusions

- SDS are a relatively mature field in terms of breadth of problems considered.
- Quite practical; FM index has been implemented in BIO software (Bowtie).
- Some foundational questions still not addressed (e.g. lower bounds). at least in dynamic SDS.

Thank You

Thank You

Special thanks to Rajeev Raman (Leicester University) and Ben Langmead (Johns Hopkins) for some of the slides