FM-index

Lecture 7

Reading: http://alexbowe.com/fm-index/

Recap: principles of indexing

- In order to perform efficient search for anything, the collection of keys to be searched should be sorted
- Suppose you want find out if string *cob* occurs as a substring of string *cocoa*.
- Because our *cocoa* is very small, it is easy to see that cob is not a part of cocoa
- Now imagine that our dataset consists of a much longer but still single non-divided sequence of characters: cocoacoacoacoacoacoa... up to 3,000,000,000 characters
- You want to know whether the *cob* substring is somewhere in this dataset

Very long strings: DNA sequences

- That reflects the typical situation with DNA sequence databases, where each sequence can be prohibitively long.
- If scanning and checking the entire dataset is out of question, you need to come up with some sort of a *cocoa* index.
- So let's treat *cocoa* as a very long string, and see how we can index this string so we can then perform search for pattern *P* in the most efficient way.

Suffix Array

Review

Suffix: reminder

- The *suffix* of a given string is a substring starting at some position and running till the end of the string.
- For example, *-coa* is a suffix of *cocoa* starting at position 2 (our position count starts at 0).

String data				
0	1	2	3	4
c	o	c	o	a

Suffix array: definition

- Take all suffixes of the input string, and sort them alphabetically (lexicographically).
- Then record a start position of each sorted suffix - and you obtain a *suffix array SA*.

String data					
0	1	2	3	4	5
c	o	c	o	a	$

Indexing *cocoa* database with suffix arrays

- First, let's add a terminal character $ with a beautiful name sentinel at the end of our cocoa: cocoa$.
- By convention, $ is lexicographically the smallest symbol in the alphabet

String data					
0	1	2	3	4	5
c	o	c	o	a	$

Suffix array of *cocoa$*

- The collection of sorted suffixes becomes:

Suffix array (SA) index		
Row	Sorted suffixes	SA (start positions)
0	$	5
1	a$	4
2	coa$	2
3	cocoa$	0
4	oa$	3
5	ocoa$	1

Only the last column is called a suffix array

Query in *log N* time

- If the entire size of the database is *N* characters (*N*=6 for *cocoa$*), we can now search for any query substring in time O(*log N*)*, using a binary search through these sorted suffixes.
- The space for this index is linear in *N*, because we store only start positions - one per character.
- Note that we need to ensure that we can easily access the original string at any random position, so we can compare our query pattern to the actual string.

More precisely: O((log N +k) M), where M is the size of the pattern and k is the number of occurrences, but we assume that M << N, and ignore that, and to find k occurrences you cannot go lower than k

Better idea:
Burrows-Wheeler Transform

The Burrows Wheeler Transform also indexes all substrings of a single string database, but in a different rather ingenious way.

Burrows-Wheeler Matrix

Circular strings

- We are back to our *cocoa$* dataset.

- First let's make our string circular: *cocoa$cocoa$...*
- That means that after we reach sentinel *$*, we start from the beginning of the never-ending string
- From this circular *cocoa$*, let's create *N* different permuted strings of length N starting at each of N possible positions: cocoa$, ocoa$c, coaco, oacoc, a$coco, $cocoa.
- Next, let's sort these strings lexicographically

Sorted circular permutations

String data					
0	1	2	3	4	5
c	o	c	o	a	$

Circular strings sorted		
Row	Sorted circular strings	Sorted suffixes
0	*$cocoa*	*$*
1	*a$coco*	*a$*
2	*coa$co*	*coa$*
3	*cocoa$*	*cocoa$*
4	*oa$coc*	*oa$*
5	*ocoa$c*	*ocoa$*

Sorted circular permutations

- Note that the order of these sorted circular strings, with regard to their start positions, is exactly the same as in the suffix array
- Why? Because after we added the unique sentinel symbol at the end of *cocoa*, there cannot be two circular strings with the same lexicographical rank (our sorted list has no ties).
- That means that the order of circular strings is uniquely determined by the part which comes **before** and includes sentinel $.
- But this is exactly the order of the suffixes - suffixes run up-to and including the sentinel!

Circular strings sorted		
Row	Sorted circular strings	Sorted suffixes
0	*$cocoa*	*$*
1	*a$coco*	*a$*
2	*coa$co*	*coa$*
3	*cocoa$*	*cocoa$*
4	*oa$coc*	*oa$*
5	*ocoa$c*	*ocoa$*

Burrows-Wheeler Matrix

Burrows-Wheeler Matrix						
Row	F	1	2	3	4	L
0	*$*	*c*	*o*	*c*	*o*	*a*
1	*a*	*$*	*c*	*o*	*c*	*o*
2	*c*	*o*	*a*	*$*	*c*	*o*
3	*c*	*o*	*c*	*o*	*a*	*$*
4	*o*	*a*	*$*	*c*	*o*	*c*
5	*o*	*c*	*o*	*a*	*$*	*c*

F and L

- Now the fun part begins.
- We concentrate only on the table of sorted circular strings, and in particular on its first and last columns: F and L:

Burrows-Wheeler Matrix						
Row	F	1	2	3	4	L
0	*$*	*c*	*o*	*c*	*o*	*a*
1	*a*	*$*	*c*	*o*	*c*	*o*
2	*c*	*o*	*a*	*$*	*c*	*o*
3	*c*	*o*	*c*	*o*	*a*	*$*
4	*o*	*a*	*$*	*c*	*o*	*c*
5	*o*	*c*	*o*	*a*	*$*	*c*

Burrows-Wheeler Transform

Burrows-Wheeler Transform (BWT)

- In the **Last column** of the Burrows-Wheeler Matrix (BWM) we see the characters preceding each sorted suffix in the suffix array, except that we made our cocoa$ circular and thus before character at position zero there is the sentinel (going around).
- The last column *L* of the BWM is the famous ***Burrows-Wheeler Transform*** (BWT).
- If we only store this last column, it occupies space not larger than the original string
- It often occupies much less space because the characters in BWT tend to be grouped into runs of equal characters, and as such the BWT string can be "compressed"

BWT is a self-index

Burrows-Wheeler Matrix						
Row	F	1	2	3	4	L
0	*$*	*c*	*o*	*c*	*o*	*a*
						o
						o
						$
4	*o*	*a*	*$*	*c*	*o*	*c*
5	*o*	*c*	*o*	*a*	*$*	*c*

If we store only the permuted string of BWT (column L), we do not actually need our original string anymore, because **we can always reconstruct the original string from the BWT string**.

Reminder

- The sequence of characters in the last row L of the Burrows-Wheeler Matrix represents a Burrows-Wheeler Transform (BWT) of the original cocoa$.
- On the other hand, this is nothing else but the sequence of characters preceding sorted suffixes in the suffix array.

Ranking characters

- Let's give each character in the first column F a rank: c with rank 1, c with rank 2 etc., in the order in which they appear in the first column of BWM.
- Obviously, each character of the alphabet appears consecutively in the first column, because it represents the first letter of lexicographically sorted suffixes.

As we mentioned before, there are no ties in the sorting of circular strings permuted from cocoa$, due to the unique sentinel added at the end, so the rank attached to each character of cocoa$ is unique.

Characters with ranks

Burrows-Wheeler Matrix						
Row	F	1	2	3	4	L
0	$1	c	o	c	o	a1
1	a1	$	c	o	c	o1
2	c1	o	a	$	c	o2
3	c2	o	c	o	a	$1
4	o1	a	$	c	o	c1
5	o2	c	o	a	$	c2

Correspondence of ranks in F and L

If we rank characters in the order in which they appear in the last column L, we notice that the ranked characters of L correspond exactly to the ranked characters of F

Burrows-Wheeler Matrix						
Row	F	1	2	3	4	L
0	$1	c	o	c	o	a1
1	a1	$	c	o	c	o1
2	c1	o	a	$	c	o2
3	c2	o	c	o	a	$1
4	o1	a	$	c	o	c1
5	o2	c	o	a	$	c2

For example, c2 in L is the character before suffix ocoa$, and in fact c2 in row 3 of column F is the first character of suffix cocoa$.

Correspondence of ranks in F and L

- The ranked characters in F and L columns always refer to the same characters of the original string.
- Why? Because F corresponds to the first letter of the alphabetically sorted suffixes, and L corresponds to the characters preceding alphabetically sorted suffixes, so if we have for example two c-s (not necessarily sequential) in L they in fact correspond to sorted suffixes c1... and c2... and thus the ranks for each letter in F and L are in exactly the same order.

LF mapping

- We stated that having only BWT (column L) we can reconstruct the original string.
- All we need for this is the BWT string itself with ranked characters and the LF-mapping table
- LF table tells, for each letter of the alphabet, the interval for all sorted suffixes which start with this letter (consecutive intervals in column F).

That is why it is called LF-mapping - because it maps ranks in the Last column to the ranks in the First column.

LF mapping table

LF-mapping table	
All suffixes starting with:	are at positions (in suffix array):
$	[0-0]
a	[1-1]
c	[2-3]
o	[4-5]

Reconstruction of the original string: 1/6 - backwards, starting with row 0

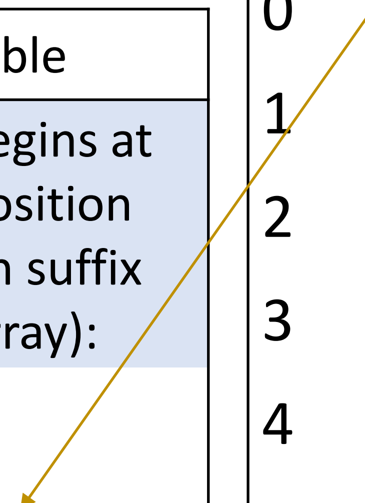
- We reconstruct the original string backwards.
- We know that the first in the SA of sorted suffixes is always the sentinel character *$*, so we trivially conclude that the last character of the original string is *$* (as if we did not add it ourselves).
- So for now we know that our original string is ***xxxxx$***.

BWT	
Row	L
0	a1
1	o1
2	o2
3	$1
4	c1
5	c2

Reconstruction of the original string: 2/6

- Which letter is before *$*? We look at row 0 of column *L* and discover that before *$* it was *a*1.
- In what row of the suffix array is the first *a*? It is in row 1, according to the *LF* table.
- So our partly recovered original string becomes ***xxxxa$***, and in search for the next (preceding) letter we jump to row 1 of column L.

LF-mapping table	
Interval of all suffixes starting with:	begins at position (in suffix array):
$	0
a	1
c	2
o	4

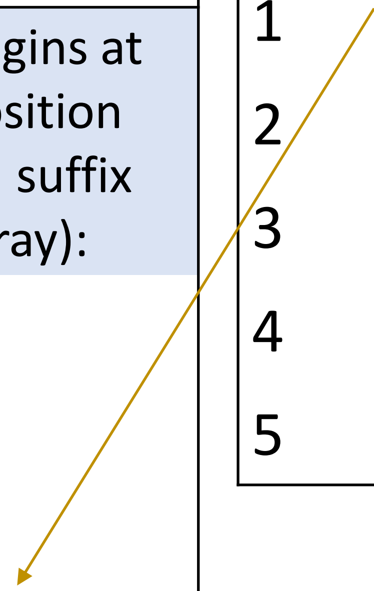


BWT	
Row	L
0	a1
1	o1
2	o2
3	$1
4	c1
5	c2

Reconstruction of the original string: 3/6

- Here, in row 1 of column L, we see that the previous letter was *o*1.
- The LF-table tells us that *o*1 (the first among all *o*-s) is the starting character of the suffix in row 4.
- We update our recovering string to *xxx**oa$***, and jump to row 4.

LF-mapping table	
Interval of all suffixes starting with:	begins at position (in suffix array):
$	0
a	1
c	2
o	4



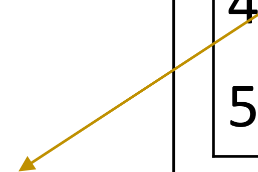
BWT	
Row	L
0	a1
1	o1
2	o2
3	$1
4	c1
5	c2

Reconstruction of the original string: 4/6

- From here we jump to *c*1 and our recovering string becomes *xx***_coa$_**.
- *c*1 corresponds to the suffix in row 2, according to the *LF*-table, so our next stop is in row 2 of column L.

LF-mapping table

Interval of all suffixes starting with:	begins at position (in suffix array):
$	0
a	1
c	2
o	4



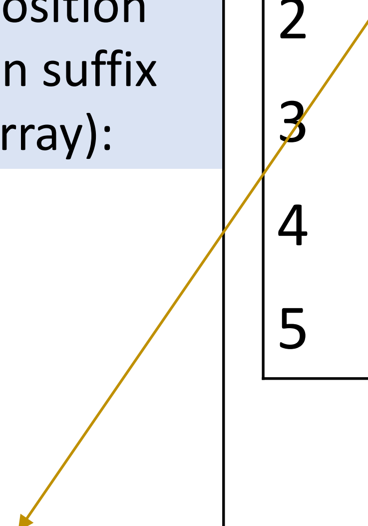
BWT

Row	L
0	a1
1	o1
2	o2
3	$1
4	c1
5	c2

Reconstruction of the original string: 5/6

- Here we see the sign: go to *o*2.
- We determine that *o*2 is in row 5 (we adding 1 to the start interval of all letters *o* in the LF-table).
- Our growing string becomes *x**ocoa$***, and we go to row 5.

LF-mapping table	
Interval of all suffixes starting with:	begins at position (in suffix array):
$	0
a	1
c	2
o	4

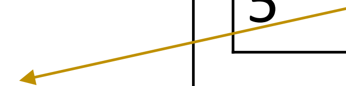


BWT	
Row	L
0	a1
1	o1
2	o2
3	$1
4	c1
5	c2

Reconstruction of the original string: 6/6

- In row 5 we see the hint: go to *c*2, which is in row 3 according to the *LF*-table.
- Our string becomes the perfect ***cocoa$***, and the sentinel in row 3 of column L signifies that the tour is over.

LF-mapping table	
Interval of all suffixes starting with:	begins at position (in suffix array):
$	0
a	1
c	2
o	4



BWT	
Row	L
0	a1
1	o1
2	o2
3	$1
4	c1
5	c2

Summary

- It means that having only a BWT string and a small LF-table, we do not need to store the original string anymore, as it can always be easily reconstructed whenever needed
- In a similar spirit we can use BWT for pattern search

That makes it an index.

In 2005 Paolo **F**erragina and Giovanni **M**anzini proposed to use the BWT with LF-table as an index for pattern search, and called the new structure the **FM**-index.

FM-index

FM-index consists of 2 tables

BWT (only L-column)			
Row	F	L	SA
0	$1	a1	5
1	a1	o1	4
2	c1	o2	2
3	c2	$1	0
4	o1	c1	3
5	o2	c2	1

LF-mapping table	
Interval of all suffixes starting with:	begins at position (in suffix array):
$	0
a	1
c	2
o	4

Existence query with FM index: 1/4

- Let's search for pattern *oco*.
- As in case of reconstructing the original string, the search proceeds backwards.
- First, we find an interval of all suffixes which start with *o*, to locate query string ***xxo***.
- This is easy: we know it from the LF-table: it tells us that these are suffixes in rows [4-5] of the suffix array.

LF-mapping table	
All chars	begin at
$	0
a	1
c	2
o	**4**

BWT (only L-column)		
Row	L	SA
0	a1	5
1	o1	4
2	o2	2
3	$1	0
4	c1	3
5	c2	1

Existence query with FM index: 2/4

- Among these suffixes, we want to know which ones are preceded with c.
- We scan rows 4-5 of the L-column, and see that both suffixes are preceded with c - with c1 to c2.
- So where is the interval in the SA that corresponds to [c1 - c2]?
- It is interval [2-3] according to the LF-table.

LF-mapping table

All chars	begin at
$	0
a	1
c	**2**
o	4

BWT (only L-column)

Row	L	SA
0	a1	5
1	o1	4
2	o2	2
3	$1	0
4	c1	3
5	c2	1

Existence query with FM index: 3/4

- So we successfully located all suffixes that start with *co*, resolving by this two last characters of the query: ***xco***

- Among all suffixes that begin with *co*, which ones are preceded with o?
- There is only one such suffix, and the preceding character is o2.
- Where is o2 located?

LF-mapping table	
All chars	begin at
$	0
a	1
c	2
o	4

BWT (only L-column)		
Row	L	SA
0	a1	5
1	o1	4
2	**o2**	2
3	$1	0
4	c1	3
5	c2	1

Existence query with FM index: 4/4

- It is located at row 5 of the suffix array, because, according to the LF-table, all *o*-suffixes start in row 4, and *o*2 suffix is the second one, that is in row 5.
- We successfully located the entire query pattern ***oco***: all the suffixes that start with *oco* are in the row 5 of the suffix array.

LF-mapping table	
All chars	begin at
$	0
a	1
c	2
o	4

BWT (only L-column)		
Row	L	SA
0	a1	5
1	o1	4
2	o2	2
3	$1	0
4	c1	3
5	c2	1

Position queries

- By using only a transformed input string and a small LF table, we could answer an existence query: yes, our dataset contains substring *oco*.
- The amazing thing is that we found the answer using the BWT string alone without scanning the entire input string and without reconstructing the original string.
- In order to find at which position(s) the query occurs, we need to look at the value of the suffix array in row 5.
- This value points to the position 1 in the original string, and thus pattern *oco* occurs at position 1 in the cocoa$ dataset.

Self-exercise

- To make it stick, try now to locate, following the same logic, pattern *coc*.

LF-mapping table	
All chars	begin at
$	0
a	1
c	2
o	4

BWT (only L-column)		
Row	L	SA
0	a1	5
1	o1	4
2	o2	2
3	$1	0
4	c1	3
5	c2	1

When the substring query does not exist: example

- What happens if we search for pattern *aoa*.
- All suffixes that start with *a* are in row 1.
- Among these (there is only one in this tiny dataset), there is one suffix that is preceded by *o*, and it refers to *o*1.
- Going to row 4 for *o*1, we are looking for all suffixes preceded by *a*.

- And there are no such suffixes!
- That means that pattern *aoa* is not present in our dataset.

Complications

- When we located an interval in the middle of the search, we need to know the smallest and the biggest rank of the next (preceding) query character.
- If this interval is very large (and it is mostly the case), how do we find the required ranks quickly?

FM-index:
Optimal search

Efficiency

- So far our search was exciting, but not terribly efficient: each time we found the suffixes in the required interval, we had to scan this interval in the L column to find the desired ranked letters for the next interval.
- The example with cocoa$ database was tiny, but imagine that we need to scan intervals of millions characters at each step of the search algorithm.

Optimal search

- To make search for pattern P of length $|P|$ to be completed in exactly $|P|$ steps we need to elaborate a little bit more on our index.

- Note that searching in $|P|$ steps is **optimal**: in order to search for pattern of length $|P|$ we have at least to read the entire pattern, which takes $|P|$ steps anyway.

- To achieve this optimal result, we enhance our index with more information.

Enhancement

- For each row, instead of storing in column L characters with ranks, we store several columns of ranks - one column for each letter of the alphabet.
- In each cell of this column we store the number of times that this particular character has been seen in column L before the position specified by the row number.
- This is nothing else but the highest rank of each character seen so far.

Extended FM-index

Extended index							
Row	F	L	rank *$*	rank *a*	rank *c*	rank *o*	SA
0	$1	a1	0	1	0	0	5
1	a1	o1	0	1	0	1	4
2	c1	o2	0	1	0	2	2
3	c2	$1	1	1	0	2	0
4	o1	c1	1	1	1	2	3
5	o2	c2	1	1	2	2	1

LF-mapping table	
All chars	begin at
$	0
a	1
c	2
o	4

Optimal search in 3 steps

- Now, let's perform the search for *oco* again.
- Because this pattern has length |P|=3, we want to perform the entire search procedure in three steps.

- Our goal is to obtain a consecutive interval in the suffix array for all suffixes that start with *oco*.

Optimal search: initialization

- To initialize, we consider the entire interval [0,5] as a starting interval.
  - start s=0,
  - end e=N=5
- As before, we search backwards.

Extended index					
Row	rank *$*	rank *a*	rank *c*	rank *o*	SA
0	0	1	0	0	5
1	0	1	0	1	4
2	0	1	0	2	2
3	1	1	0	2	0
4	1	1	1	2	3
5	1	1	2	2	1

LF-mappin table	
All chars	be at
$	0
a	1
c	2
o	4

Optimal search: step 1/3

- In the interval [s,e] we search for suffixes that start with *o*.

In row s=0 the rank (at pos s=0, of o) = 0

In row e = 5 rank (at pos e=5, of o) = 2

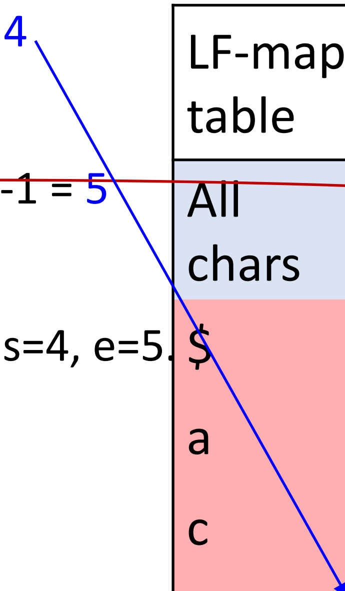
- If the rank is zero, then this means that all preceding o's in this interval start directly from the first position of o's in the LF-table: position 4
- The end of the interval is:

LF(o) + rank (at pos e=5, of *o*) -1 = 5

- The new interval becomes: s=4, e=5.

- Result so far: [4,5].

Extended index					
Row	rank *$*	rank *a*	rank *c*	rank *o*	SA
0	0	1	0	0	5
1	0	1	0	1	4
2	0	1	0	2	2
3	1	1	0	2	0
4	1	1	1	2	3
5	1	1	2	2	1



LF-mapping table	
All chars	be at
$	0
a	1
c	2
o	4

Optimal search: step 2/3

- In the interval [4,5] what are the suffixes that have a letter *c* before *o*?

- To answer this, we look at rank (at pos s=4, of *c*) and rank (at pos e=5, of *c*):

rank(4, c) = 1

rank (5, c) = 2

- That means that the preceding *c*'s are between the first and the second *c* in the suffix array.

LF-mappin table

All chars	be at
$	0
a	1
c	2
o	4

Extended index

Row	rank *$*	rank *a*	rank *c*	rank *o*	SA
0	0	1	0	0	5
1	0	1	0	1	4
2	0	1	0	2	2
3	1	1	0	2	0
4	1	1	1	2	3
5	1	1	2	2	1

Optimal search: step 2/3

- The start and end of a new interval is computed by:

s = LF (c) + rank(4,c) - 1 =
2 + 1 -1=2

e = LF (c) + rank (5,c) - 1=
2 + 2 - 1=3

- And the narrowed interval after step 2 becomes: [2, 3].

LF-mapping table

All chars	be at
$	0
a	1
c	2
o	4

Extended index

Row	rank *$*	rank *a*	rank *c*	rank *o*	SA
0	0	1	0	0	5
1	0	1	0	1	4
2	0	1	0	2	2
3	1	1	0	2	0
4	1	1	1	2	3
5	1	1	2	2	1

Optimal search: step 3/3

- In the interval [2,3] what are the suffixes that are preceded by *o*?
- The rank of *o* at position 2 is 2, and the rank of *o* at position 3 is 2, so the new values of start and end become:

s = LF (o) + rank(2,o) -1

= 4 +2 -1 = 5

e = LF (o) + rank (3,o) - 1

= 4 + 2 -1 = 5

- Our final interval is [5,5].

LF-mapping table

All chars	be at
$	0
a	1
c	2
o	4

Extended index

Row	rank *$*	rank *a*	rank *c*	rank *o*	SA
0	0	1	0	0	5
1	0	1	0	1	4
2	0	1	0	2	2
3	1	1	0	2	0
4	1	1	1	2	3
5	1	1	2	2	1

General formula for the next interval

- To compute the next start position for the interval of all suffixes starting with character c from the current start row s:

c_rank = rank (c in row s)

if c_rank > 0 c_rank = c_rank -1

next_s = LF (c) + c_rank

- To compute the next end position for the interval of all suffixes starting with character c from the current end row e:

c_rank = rank (c in row e)

if c_rank = 0: pattern does not occur in the input string

else: c_rank = c_rank -1

next_e = LF (c) + c_rank

This is an amazing result: not only we can compress our input string, but we can search for pattern P in optimal time $O(|P|)$, without decompression.