Exercises

1) For each mapping approach:

   i) Why not use the naive, brute force sliding window approach? time

   ii) Why not use hashes? memory, variable length strings

   iii) Why not use suffix trees? memory

   iv) Why not use suffix arrays? memory

2) Why are data structures important for the string search problem? affects time and memory

3) Which data structure would you use to search for substrings of varying lengths? suffix trees or suffix arrays, depending on memory constraints

Today's Objectives

- Evaluating strategies for short read alignment
  - String search
  - Linear (time constraints)
  - Hash
  - Suffix tree
  - Suffix array
  - Burrows Wheeler Transform (BWT)
  - FM index search of BWT

memory constraints

Burrows-Wheeler Transform (BWT)

T=abaaba$

Rotation	Sort (BW matrix)
abaaba$	**$**abaaba
baaba$a	**a$**abaab
aaba$ab	**aaba$**ab
aba$aba	**aba$**aba
ba$abaa	**abaaba$**
a$abaab	**ba$**abaa
$abaaba	**baaba$**a

Suffix Array

Last column

BWT(T)=
abba$aa

BWT allows Lossless compression - the original data can be perfectly reconstructed from the compressed data.
BWT is reversible
'Compressed Suffix Array'

BWT enables compression

How is it compressed?

```
BWT("Tomorrow_and_tomorrow_and_tomorrow$")
='w$wwdd__nnoooaattTmmmrrrrrrooo__ooo'
```

BWT (also called block-sorting compression) rearranges a character string into runs of similar characters. This is useful for compression, since it tends to be easy to compress a string that has runs of repeated characters by techniques such as move-to-front transform and run-length encoding.

The transformation is reversible. Lossless compression is a class of data compression algorithms that allows the original data to be perfectly reconstructed from the compressed data. Lossy compression only allows approximate reconstruction of the data

Lossless compression algorithms: ZIP, GNU tool gzip, PNG and GIF

Compression

Run length encoding

runs of data are stored as a single data value and count

AAAAAATTTTTTTCTTTTTTGGGG

6A7T1C6T4G

Arithmetic coding

string of characters such as the word "java" is represented using a fixed number of bits per character

j = 01, a = 11, v = 10

java = 01111011

1 character = 8 bits (1 byte)

By using smaller bit codes for common words/characters, we can get compression.

Huffman coding

Char	Code	Freq	Bits
E	10	15	30
P	01	13	26
A	110	10	30
S	11111	3	15

BWT

How is it an index, or how do we reverse it?

$	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

BWM(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 $

SA(T)

T=abaaba$

BWT(T)=
abba$aa

$abaaba
a$abaab
aaba$ab
aba$aba
abaaba$
ba$abaa
baaba$a

Reversing BWT(T)

BWT(T)	Sort	Prepend BWT(T)	Sort	Prepend BWT(T)
a	$	a$	$a	a$a
b	a	ba	a$	ba$
b	a	ba	aa	baa
a	a	aa	ab	aab
$	a	$a	ab	$ab
a	b	ab	ba	aba
a	b	ab	ba	aba

$aba
a$ab
aaba
aba$
abaa
ba$a
baab

Sort

a$aba
ba$ab
baaba
aaba$
$abaa
aba$a
abaab

Prepend

$abaa
a$aba
aaba$
aba$a
abaab
ba$ab
baaba

Sort

a$abaa
ba$aba
baaba$
aaba$a
$abaab
aba$ab
abaaba

Prepend

$abaab
a$abaa
aaba$a
aba$ab
abaaba
ba$aba
baaba$

Sort

a$abaab
ba$abaa
baaba$a
aaba$ab
$abaaba
aba$aba
abaaba$

Prepend

$abaaba
a$abaab
aaba$ab
aba$aba
abaaba$
ba$abaa
baaba$a

Sort

T=abaaba$

BWT(T)=
abba$aa

Seems like a lot of computing, is there an easier way?

$abaaba
a$abaab
aaba$ab
aba$aba
abaaba$
ba$abaa
baaba$a

Reversing BWT with FM index and LF mapping

FM index (Full test Minute space)

T-ranking

Give each character in T a rank, equal to # times the character occurred previously in T. (see structure)

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

B-ranking

Give each character in BWT(T) a rank, equal to # times the character occurred. (Use for LF mapping)

T-ranking

T-ranking

Give each character in T a rank, equal to # times the character occurred previously in T.

a_0 b_0 a_1 a_2 b_1 a_3 $

$ abaaba$_3$
$a_3$$abaab$_1$
$a_1$abaab_0$
$a_2$baaba_1$
$a_0$baaba$
b_1a$abaa$_2$
$b_0$aaba$a$_0$

BWM with T-ranking

$	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

BWM(T)

First						Last
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

BWM(T) with T-ranking

Order does not change!

BWM with B-ranking

B-ranking

Give each character in BWT(T) a rank, equal to # times the character occurred.

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

BWM(T) with T-ranking

Order does not change!

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

BWM(T) with B-ranking

Order of a_i and b_i does not change!

Last First (LF) mapping
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

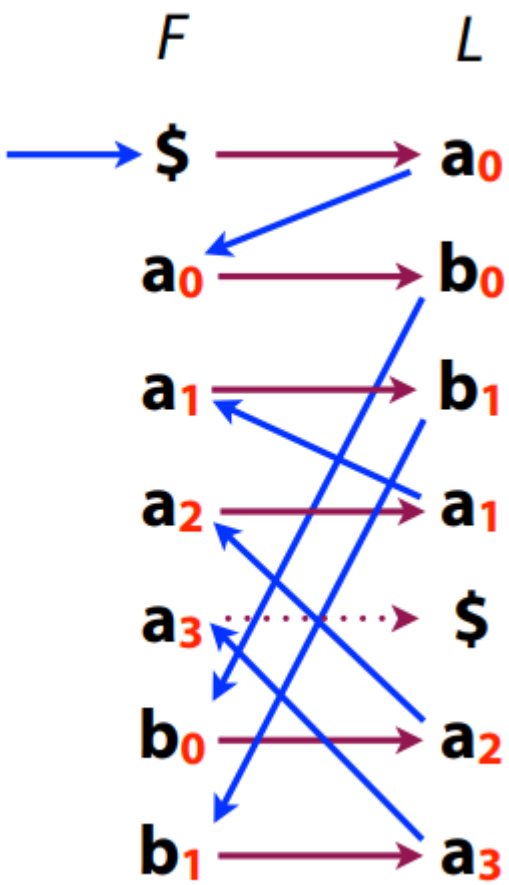
BWM(T) with B-ranking

F can be represented by n integers, n = alphabet size
4 (a), 2 (b)

- To start at the end ($) we go to the first position in (F).
- What is before $? Go to same position in (L) a_0
- What is before a_0? Find the row that begins with a_0. Skip $ when counting.
- b_0 is before a_0
- What is before b_0? Skip $ (1), a (4)
- a_2 is before b_0

	F	L	
	$	a_0	
	a_0	b_0	
	a_1	b_1	← Which BWM row *begins* with b_1?
	a_2	a_1	Skip row starting with $ (1 row)
	a_3	$	Skip rows starting with a (4 rows)
	b_0	a_2	Skip row starting with b_0 (1 row)
row 6 →	b_1	a_3	Answer: row 6

Reverse BWT



Given L, and positions of F rows, we can reverse BW(T) to T.

Reverse of chars we visited = $\mathbf{a_3\ b_1\ a_1\ a_2\ b_0\ a_0\ \$} = T$

Storage

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

FM-index is a compressed full-text substring index based on the Burrows-Wheeler transform

- F can be represented by n integers, n = alphabet size
- L can be compressed

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 $

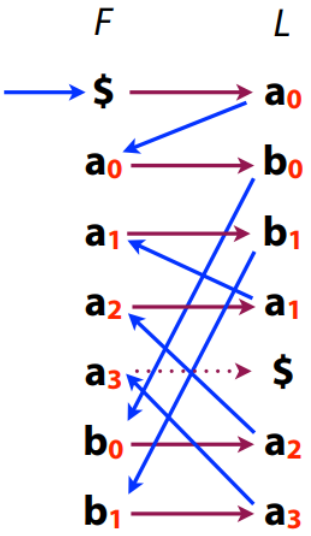
Suffix Array

- Array of integers
- Array of characters
- no compression

Exercises

1) Construct BWT(T), with T = 'CATGCAT'

2) Draw out LF mapping to show how to reverse BWT(T) using arrows and LF columns as shown on the right:



3) Given BWT(T) = ATGCA$A, what is F, draw LF mapping and give T.

FM index search of BWT

FM index Query

Search the reverse of the string

$P = \mathbf{aba}$

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 Query

$P = \mathbf{aba}$

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 →

$P = \mathbf{aba}$

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 Query

$P = \mathbf{aba}$

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 →

$P = \mathbf{aba}$

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**

Location of matches

	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
[3, 5)	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 are these?

	6	$
	5	a $
	2	a a b a $
[3, 5)	3	a b a $
	0	a b a a b a $
	4	b a $
	1	b a a b a $

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

Issues to resolve

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

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
Where?	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

Storing ranks takes too much space

Pre-tally ranks

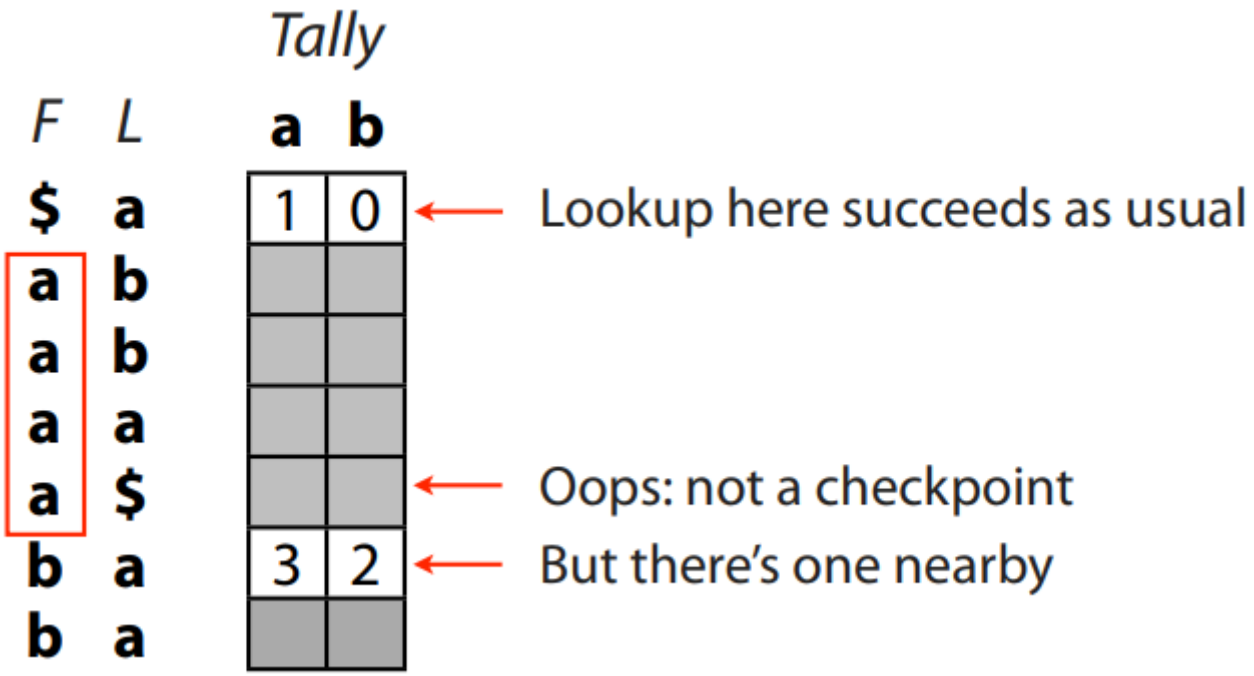
Idea: pre-calculate # **a**s, **b**s in *L* up to every row:

F	*L*	*Tally* **a**	*Tally* **b**
$	**a**	1	0
a	**b**	1	1
a	**b**	1	2
a	**a**	2	2
a	**$**	2	2
b	**a**	3	2
b	**a**	4	2

We infer $\mathbf{b}_0$ and $\mathbf{b}_1$ appear in *L* in this range

O(1) time, but requires $m \times |\Sigma|$ integers

Exchange memory for speed using checkpoints



What's my rank?

482 + 2 - 1 = 483

Checkpoint

as along the way

tally -> rank

What's my rank?

439 - 2 - 1 = 436

	Tally	
L	**a**	**b**
⋮	⋮	
a	482	432
b		
b		
a		
a		
a		
a		
b		
b		
b		
a		
a		
b		
b	488	439
a		

Assuming checkpoints are spaced $O(1)$ distance apart, lookups are $O(1)$

```
letter: aaaa
Tally:  1234
Rank:   0123
```

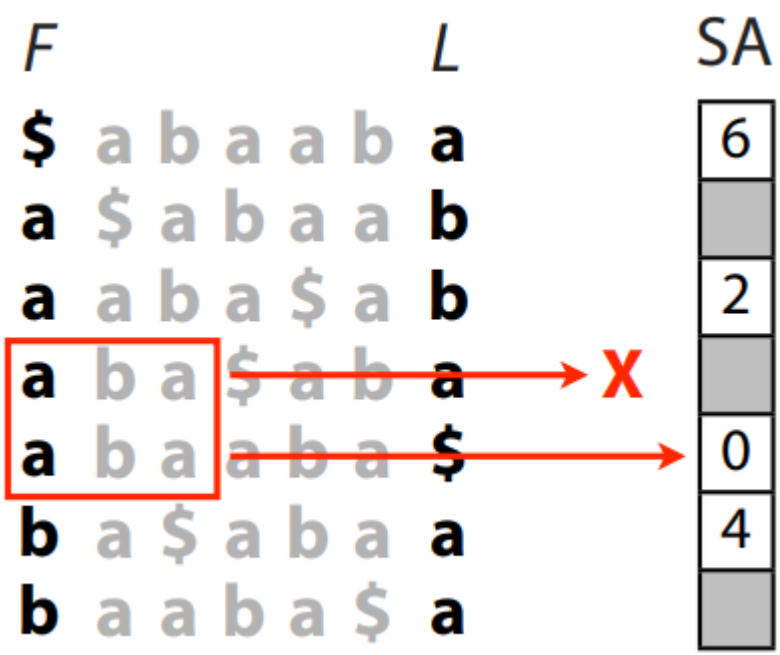
	Tally	
L	**a**	**b**
⋮	⋮	
a	482	432
b		
b		
a		
a		
a		
a		
b		
b		
b		
a		
a		
b		
b	488	439
a		

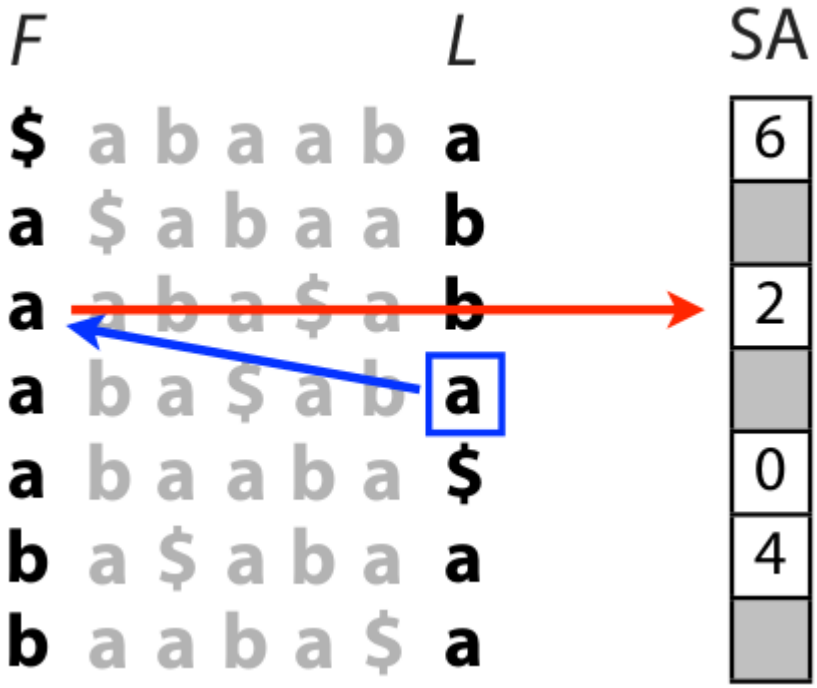
Given this range, find the tally of 'a': 483-486
What about 'b': 434-437

	Tally	
L	**a**	**b**
⋮	⋮	
a	482	432
b		
b		
a		
a		
a		
a		
b		
b		
b		
a		
a		
b		
b	488	439
a		

Given this range, find the tally of 'a': 483-486
What about 'b': 434-437
Only need to count to the nearest checkpoint

Positions with subset of Suffix Array





If a[1] is at position 2 in SA
Then a[3] is at position 3 in SA
aba is at positions [0] and [3]

Components of the FM Index:

First column (F):	$\sim \lvert \Sigma \rvert$ integers
Last column (L):	m characters
SA sample:	$m \cdot a$ integers, where a is fraction of rows kept
Checkpoints:	$m \times \lvert \Sigma \rvert \cdot b$ integers, where b is fraction of rows checkpointed

Example: DNA alphabet (2 bits per nucleotide), T = human genome, $a = 1/32$, $b = 1/128$

First column (F):	16 bytes
Last column (L):	2 bits * 3 billion chars = 750 MB
SA sample:	3 billion chars * 4 bytes/char / 32 = ~ 400 MB
Checkpoints:	*3 billion* * 4 bytes/char / *128* = ~ 100 MB
	Total < 1.5 GB

Problems and solutions

We now have fast, memory efficient string search

What if our reads are not a perfect match (sequencing errors or SNPs)

- Seed and extend
- Seed and align (extend with gaps)

Solution: Seeding is needed to avoid mismatches (errors & SNPs)

How many seeds? If the read has 30 characters, and seed length is 10, and the seed interval is 6, the seeds extracted will be:

```
Read:        TAGCTACGCTCTACGCTATCATGCATAAAC
Seed 1 fw:   TAGCTACGCT
Seed 1 rc:   AGCGTAGCTA
Seed 2 fw:         CGCTCTACGC
Seed 2 rc:         GCGTAGAGCG
Seed 3 fw:               ACGCTATCAT
Seed 3 rc:               ATGATAGCGT
Seed 4 fw:                     TCATGCATAA
Seed 4 rc:                     TTATGCATGA
```

Whats the largest number of mismatches that can occur and still **guarantee** at least one seed? 1

Solution: Seeding is needed to avoid mismatches (errors & SNPs)

How many seeds? If the read has 30 characters, and seed length is 10, and the seed interval is 6, the seeds extracted will be:

```
Read:        TAGCTACGCTCTACGCTATCATGCATAAAC
Seed 1 fw:   TAGCTACGCT
Seed 1 rc:   AGCGTAGCTA
Seed 2 fw:         CGCTCTACGC
Seed 2 rc:         GCGTAGAGCG
Seed 3 fw:                 ACGCTATCAT
Seed 3 rc:                 ATGATAGCGT
Seed 4 fw:                       TCATGCATAA
Seed 4 rc:                       TTATGCATGA
```

Whats the largest number of mismatches that can occur and still **guarantee** at least one seed? 1

Solution: seed and extend

Read: **TAGCTACGCTCTACGCTGTCATGCATAAAC**
Seed: **TAGCTACGCT**

Genome:

Seed

ATCGATTAGCTACGCTCTACGCTATCATGCATAAACCAGCATCGCA

Extend

Mismatches are ok

What about gaps

```
Read:  TAGCTACGCTCTACGCTGTCATGCATAAAC
Seed:  TAGCTACGCT
```

TAAAC	TAAAC	TAAAC
TACCA	T--ACCA	T-ACCA
3S	0S	1S
0G	2G	1G

Genome:

Seed

```
ATCGATTAGCTACGCTCTACGCTATCATGCATACCAGCATCGCA
```

Extend

Gaps are not ok

Exercises

1) Find positions of F for 'baa', 'aab' and 'ab' using FM index query.

F	L	Position of F
$	b	x[0]
a	b	x[1]
a	b	x[3]
a	b	x[4]
a	$	x[5]
a	a	x[6]
a	a	x[7]
a	a	x[8]
b	b	x[9]
b	a	x[10]
b	b	x[11]
b	a	x[12]
b	a	x[13]
b	a	x[14]

For example: 'ab' must start with 'a' and so could be in positions 1-8 of x (which stores positions). The question is which of these 'a' positions are followed by 'b'. Remember to search for the reverse of the string.