Combinatorial Optimization in Bioinformatics

Read Mapping / Combinatorial Pattern Matching

Sebastian Will · Yann Ponty

sebastian.will@polytechnique.edu · yann.ponty@lix.polytechnique.fr

AMIBio

Compeau, Pevzner. Bioinformatics Algorithms.
Chapter: Combinatorial Pattern Matching

How to identify the SNPs of an individual human

- start by sequencing the genome, millions of short reads

How to identify the SNPs of an individual human

- start by sequencing the genome, millions of short reads
- **assemble** the genome **(!???)**

How to identify the SNPs of an individual human

- start by sequencing the genome, millions of short reads
- **assemble** the genome **(!???)**
- instead: **map** the reads to a reference genome

How to identify the SNPs of an individual human

- start by sequencing the genome, millions of short reads
- **assemble** the genome **(!???)**
- instead: **map** the reads to a reference genome
- (SNPs are only one motivation for mapping, there are many others)

How to identify the SNPs of an individual human

- start by sequencing the genome, millions of short reads
- **assemble** the genome **(!???)**
- instead: **map** the reads to a reference genome
- (SNPs are only one motivation for mapping, there are many others)

How to identify the SNPs of an individual human

- start by sequencing the genome, millions of short reads
- **assemble** the genome **(!???)**
- instead: **map** the reads to a reference genome
- (SNPs are only one motivation for mapping, there are many others)

Stop and Think: Why should mapping be easier?

How to identify the SNPs of an individual human

- start by sequencing the genome, millions of short reads
- **assemble** the genome **(!???)**
- instead: **map** the reads to a reference genome
- (SNPs are only one motivation for mapping, there are many others)

Stop and Think: Why should mapping be easier?

Stop and Think: What computational challenges arise from mapping millions of reads to a human reference genome?

Multiple pattern matching

In text (e.g., bannamabananpananasbanana) of length n, find all occurrences of k words (e.g., { ananas, anna, banana, banama }); maximum word length m

Multiple pattern matching

In text (e.g., bannamabananpananasbanana) of length n, find all occurrences of k words (e.g., { ananas, anna, banana, banama }); maximum word length m

For each word:

```
bannamabananpananasbanana
ananas
 ananas
      ......
        ananas
           ......
             ananas
```

- search each single word by scanning through the text: $O(k \times n \times m)$ steps
- use better single search algorithm (Knuth-Morris-Pratt, Boyer-Moore): $O(k \times n)$ steps

Multiple pattern matching

In text (e.g., bannamabananpananasbanana) of length n, find all occurrences of k words (e.g., { ananas, anna, banana, banama }); maximum word length m

For each word:

```
bannamabananpananasbanana
ananas
 ananas
      ......
        ananas
           ......
             ananas
```

- search each single word by scanning through the text: $O(k \times n \times m)$ steps
- use better single search algorithm (Knuth-Morris-Pratt, Boyer-Moore): $O(k \times n)$ steps

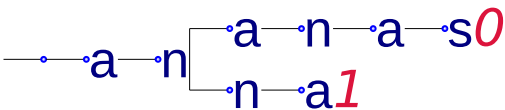
Stop and Think: Are we happy? Does it help for mapping?

Prefix tries

Construct trie from set of words (e.g., { ananas, anna, banana, banama })

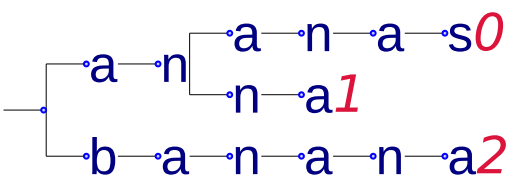
Prefix tries

Construct trie from set of words (e.g., { ananas, anna, banana, banama })



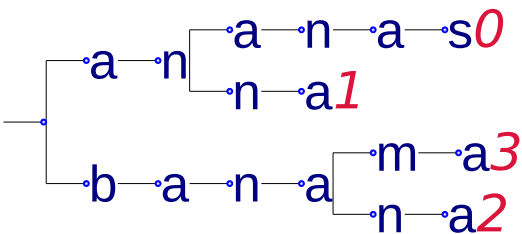
Prefix tries

Construct trie from set of words (e.g., { ananas, anna, banana, banama })



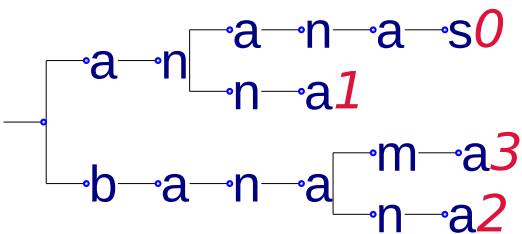
Prefix tries

Construct trie from set of words (e.g., { ananas, anna, banana, banama })



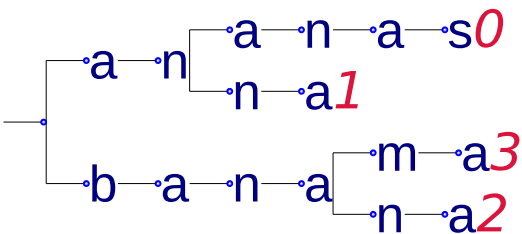
Prefix tries

Construct trie from set of words (e.g., { ananas, anna, banana, banama })



Prefix tries

Construct trie from set of words (e.g., { ananas, anna, banana, banama })

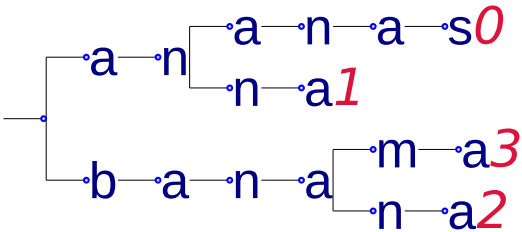


Multiple pattern search: match trie to text at each text position

bannamabananpananasbanana

Prefix tries

Construct trie from set of words (e.g., { ananas, anna, banana, banama })

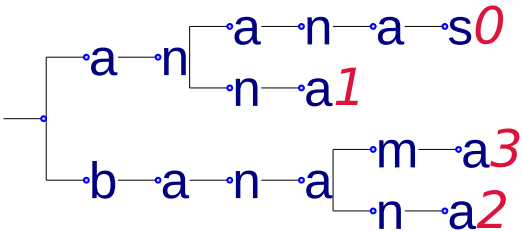


Multiple pattern search: match trie to text at each text position
bannamabananpananasbanana

Stop and Think: Where does this still perform redundant work? (solved by Aho-Corasick-Automata)

Prefix tries

Construct trie from set of words (e.g., { ananas, anna, banana, banama })

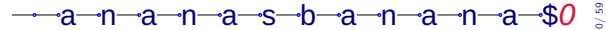


Multiple pattern search: match trie to text at each text position
bannamabananpananasbanana

Stop and Think: How does this perform for mapping?

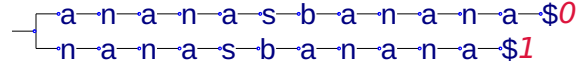
Suffix tries

Build **index for the text**, not for patterns!
Trie of all suffixes of text ananasbanana$



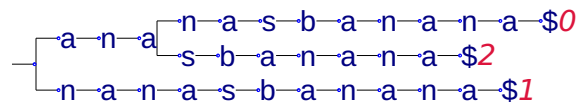
Suffix tries

Build **index for the text**, not for patterns!
Trie of all suffixes of text ananasbanana$



Suffix tries

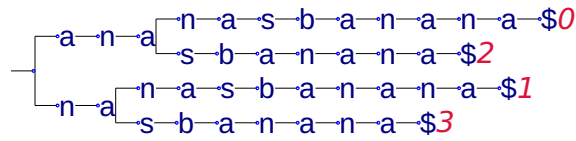
Build **index for the text**, not for patterns!
Trie of all suffixes of text ananasbanana$



Suffix tries

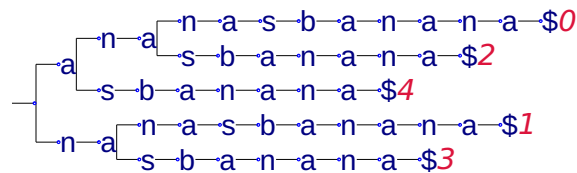
Build **index for the text**, not for patterns!

Trie of all suffixes of text ananasbanana$



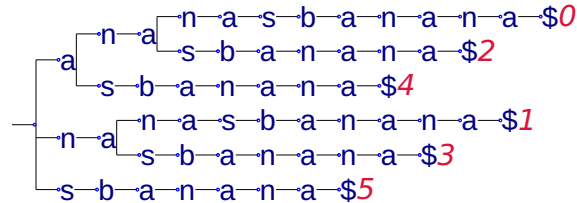
Suffix tries

Build **index for the text**, not for patterns!
Trie of all suffixes of text ananasbanana$



Suffix tries

Build **index for the text**, not for patterns!
Trie of all suffixes of text ananasbanana$

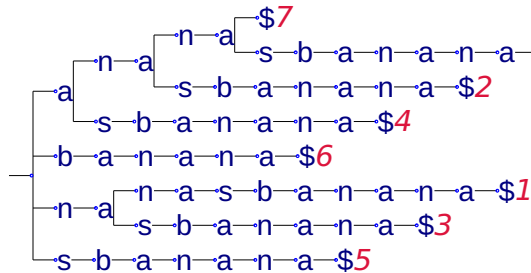


Suffix tries

Build **index for the text**, not for patterns!
Trie of all suffixes of text ananasbanana$

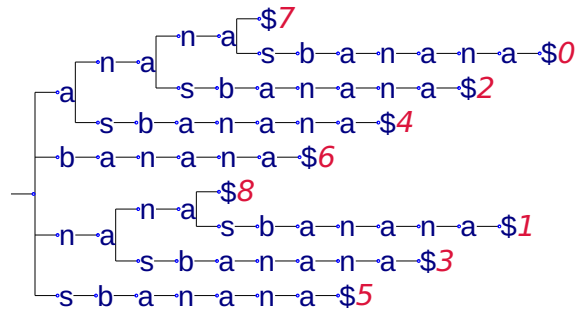
Suffix tries

Build **index for the text**, not for patterns!
Trie of all suffixes of text ananasbanana$



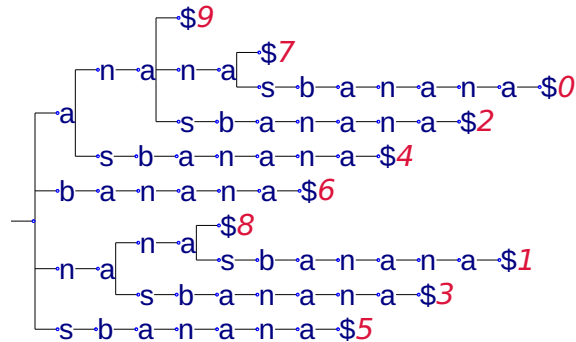
Suffix tries

Build **index for the text**, not for patterns!
Trie of all suffixes of text ananasbanana$



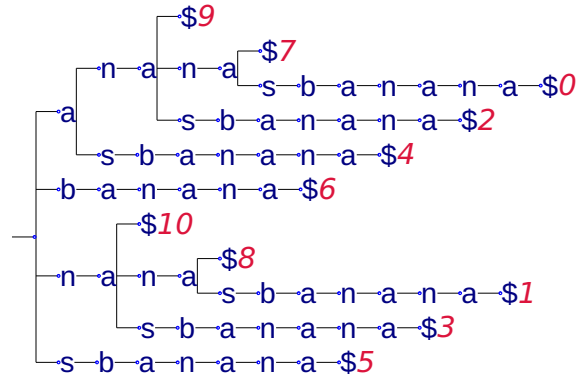
Suffix tries

Build **index for the text**, not for patterns!
Trie of all suffixes of text ananasbanana$



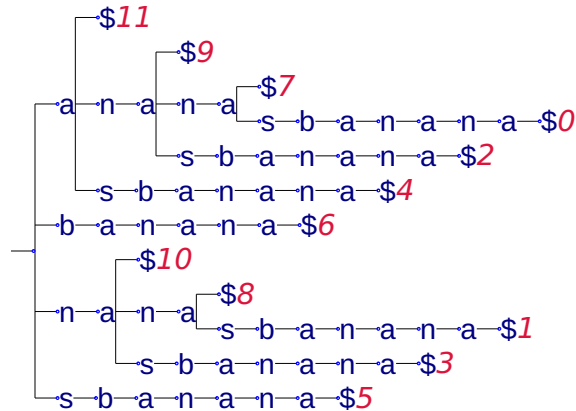
Suffix tries

Build **index for the text**, not for patterns!
Trie of all suffixes of text ananasbanana$



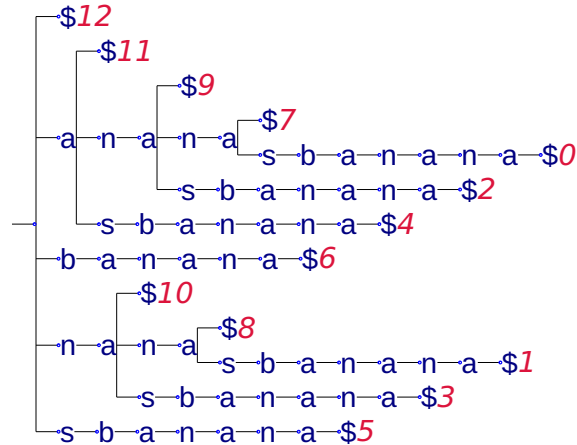
Suffix tries

Build **index for the text**, not for patterns!
Trie of all suffixes of text ananasbanana$



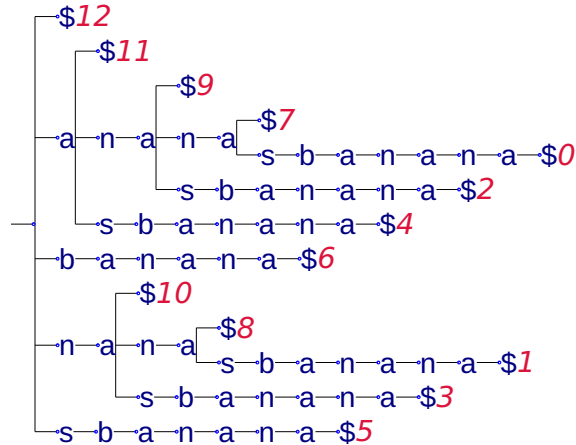
Suffix tries

Build **index for the text**, not for patterns!
Trie of all suffixes of text ananasbanana$



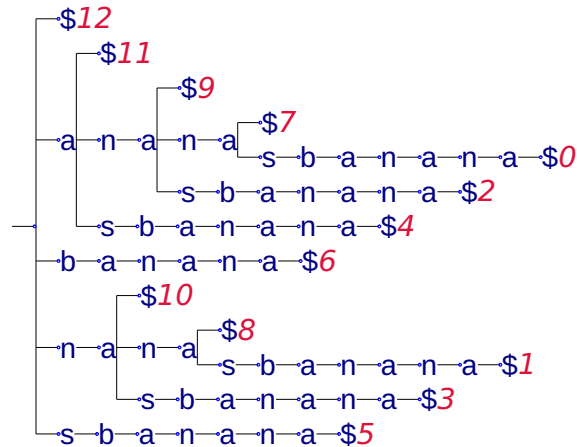
Suffix tries

Build **index for the text**, not for patterns!
Trie of all suffixes of text ananasbanana$



Suffix tries

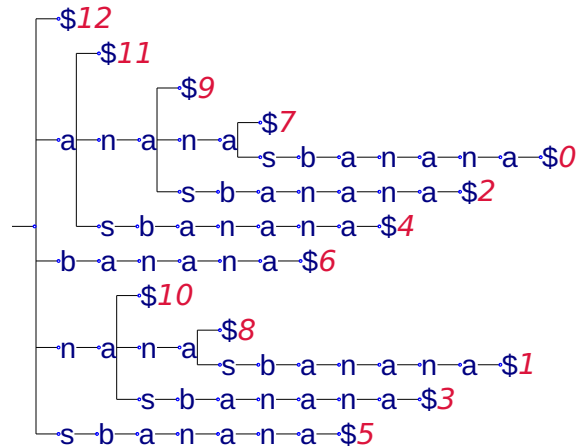
Build **index for the text**, not for patterns!
Trie of all suffixes of text ananasbanana$



Stop and Think: How do we map words / reads?
(e.g. find all occurrences of an)

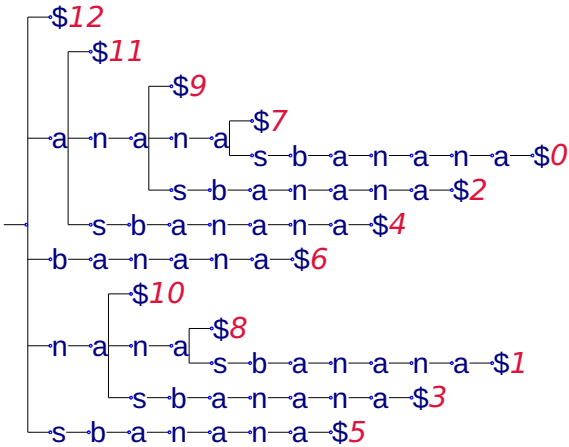
Suffix tries

Build **index for the text**, not for patterns!
Trie of all suffixes of text ananasbanana$

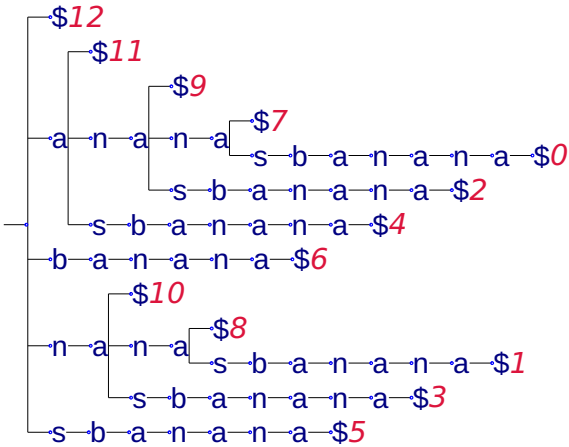


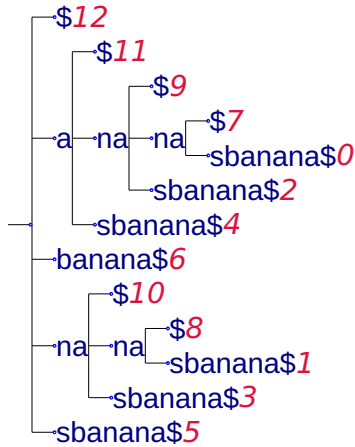
Stop and Think: What did we gain? What can we still improve? What about our intended application?

Compact suffix trees

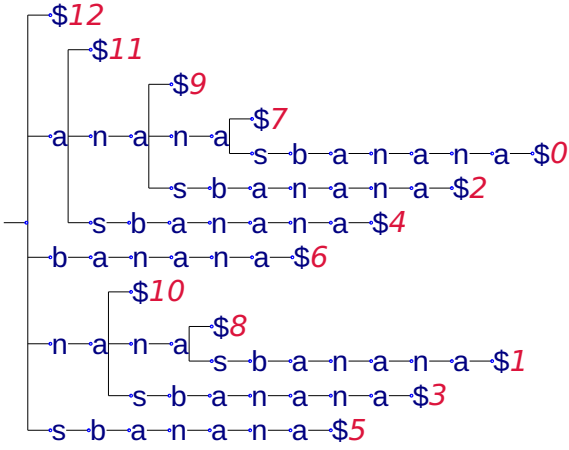


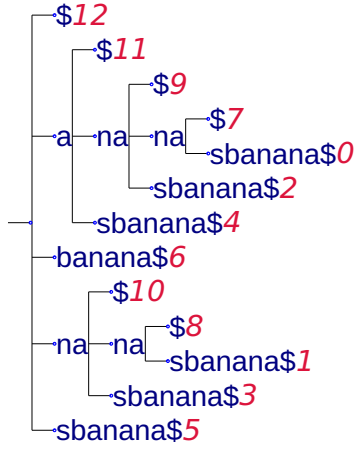
Compact suffix trees





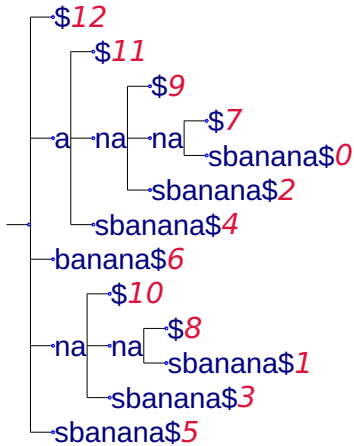
Compact suffix trees





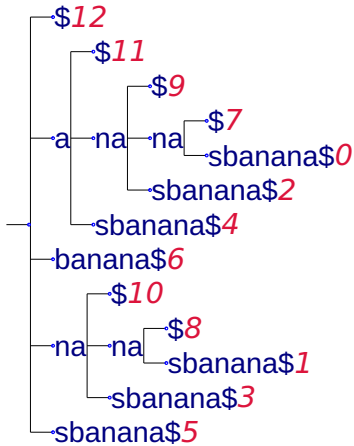
Stop and Think: **How does this change the search? What do we gain? (How to represent it in memory?) How to go on?**

Saving more space: suffix arrays



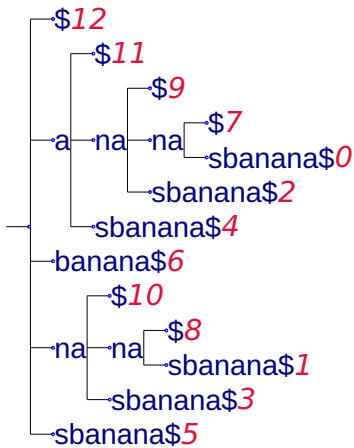
suffix	SA
$	12
a$	11
ana$	9
anana$	7
ananasbanana$	0
anasbanana$	2
asbanana$	4
banana$	6
na$	10
nana$	8
nanasbanana$	1
nasbanana$	3
sbanana$	5

Saving more space: suffix arrays



suffix	SA
$	12
a$	11
ana$	9
anana$	7
ananasbanana$	0
anasbanana$	2
asbanana$	4
banana$	6
na$	10
nana$	8
nanasbanana$	1
nasbanana$	3
sbanana$	5

Saving more space: suffix arrays



suffix	SA
$	12
a$	11
ana$	9
anana$	7
ananasbanana$	0
anasbanana$	2
asbanana$	4
banana$	6
na$	10
nana$	8
nanasbanana$	1
nasbanana$	3
sbanana$	5

Suffix array (SA) constructed from tree or sorting suffixes. Text + SA alone allows efficient mapping. Much reduced memory over suf-tree. In practice: Enhanced SA (ESA).

An index as compact as the text itself?
The Burrows-Wheeler transform

Text: ananasbanana$

rotate

```
 0 ananasbanana$
 1 nanasbanana$a
 2 anasbanana$an
 3 nasbanana$ana
 4 asbanana$anan
 5 sbanana$anana
 6 banana$ananas
 7 anana$ananasb
 8 nana$ananasba
 9 ana$ananasban
10 na$ananasbana
11 a$ananasbanan
12 $ananasbanana
```

An index as compact as the text itself?
The Burrows-Wheeler transform

Text: ananasbanana$

	0 ananasbanana$		$ananasbanana
	1 nanasbanana$a		a$ananasbanan
	2 anasbanana$an		ana$ananasban
	3 nasbanana$ana		anana$ananasb
	4 asbanana$anan		ananasbanana$
	5 sbanana$anana		anasbanana$an
rotate	6 banana$ananas	... and sort	asbanana$anan
	7 anana$ananasb		banana$ananas
	8 nana$ananasba		na$ananasbana
	9 ana$ananasban		nana$ananasba
	10 na$ananasbana		nanasbanana$a
	11 a$ananasbanan		nasbanana$ana
	12 $ananasbanana		sbanana$anana

BWT(Text): annb$nnsaaaaa

Inverting BWT

```
 0 $ananasbanana
 1 a$ananasbanan
 2 ana$ananasban
 3 anana$ananasb
 4 ananasbanana$
 5 anasbanana$an
 6 asbanana$anan
 7 banana$ananas
 8 na$ananasbana
 9 nana$ananasba
10 nanasbanana$a
11 nasbanana$ana
12 sbanana$anana
```

- Reconstruct text from its BWT **annb$nnsaaaaa**
- By sorting the BWT, we get the first column!
- Info from each row: first character is preceeded by last.
- Reconstruct from right to left:
  - from line 0, $ is preceeded by a
  - From lines 8–12: a is preceeded by n OR s
  - we need to resolve the ambiguity!

Inverting BWT

```
 0 $ananasbanana
 1 a$ananasbanan
 2 ana$ananasban
 3 anana$ananasb
 4 ananasbanana$
 5 anasbanana$an
 6 asbanana$anan
 7 banana$ananas
 8 na$ananasbana
 9 nana$ananasba
10 nanasbanana$a
11 nasbanana$ana
12 sbanana$anana
```

- Reconstruct text from its BWT **annb$nnsaaaaa**
- By sorting the BWT, we get the first column!
- Info from each row: first character is preceeded by last.
- Reconstruct from right to left:
  - from line 0, $ is preceeded by a
  - From lines 8–12: a is preceeded by n OR s
  - we need to resolve the ambiguity! DONE
  - $ is preceeded by first a (row 0)
  - first a is preceeded by first n (row 1)
  - first n $\rightarrow$ second a (row 8)
  - second a $\rightarrow$ second n (row 2)
  - ...

BWT preserves order

0	$ananasbanana
1	a$ananasbanan
2	ana$ananasban
3	anana$ananasb
4	ananasbanana$
5	anasbanana$an
6	asbanana$anan
7	banana$ananas
8	na$ananasbana
9	nana$ananasba
10	nanasbanana$a
11	nasbanana$ana
12	sbanana$anana

1	a$ananasbanan	$ananasbanana	0
2	ana$ananasban	na$ananasbana	8
3	anana$ananasb	nana$ananasba	9
4	ananasbanana$	nanasbanana$a	10
5	anasbanana$an	nasbanana$ana	11
6	asbanana$anan	sbanana$anana	12

0	$ananasbanana
1	a$ananasbanan
2	ana$ananasban
3	anana$ananasb
4	ananasbanana$
5	anasbanana$an
6	asbanana$anan
7	banana$ananas
8	na$ananasbana
9	nana$ananasba
10	nanasbanana$a
11	nasbanana$ana
12	sbanana$anana

BWT preserves order

0	$ananasbanana
1	a$ananasbanan
2	ana$ananasban
3	anana$ananasb
4	ananasbanana$
5	anasbanana$an
6	asbanana$anan
7	banana$ananas
8	na$ananasbana
9	nana$ananasba
10	nanasbanana$a
11	nasbanana$ana
12	sbanana$anana

1	a$ananasbanan
2	ana$ananasban
3	anana$ananasb
4	ananasbanana$
5	anasbanana$an
6	asbanana$anan

$ananasbanana	0
na$ananasbana	8
nana$ananasba	9
nanasbanana$a	10
nasbanana$ana	11
sbanana$anana	12

0	$ananasbanana
1	a$ananasbanan
2	ana$ananasban
3	anana$ananasb
4	ananasbanana$
5	anasbanana$an
6	asbanana$anan
7	banana$ananas
8	na$ananasbana
9	nana$ananasba
10	nanasbanana$a
11	nasbanana$ana
12	sbanana$anana

*i*th a in first and last column both correspond to the same a in the text.

BWT preserves order

0	$ananasbanana		
1	a$ananasbanan	1	a$ananasbanan
2	ana$ananasban	2	ana$ananasban
3	anana$ananasb	3	anana$ananasb
4	ananasbanana$	4	ananasbanana$
5	anasbanana$an	5	anasbanana$an
6	asbanana$anan	6	asbanana$anan
7	banana$ananas		
8	na$ananasbana		
9	nana$ananasba		
10	nanasbanana$a		
11	nasbanana$ana		
12	sbanana$anana		

$ananasbanana	0
na$ananasbana	8
nana$ananasba	9
nanasbanana$a	10
nasbanana$ana	11
sbanana$anana	12

0	$ananasbanana
1	a$ananasbanan
2	ana$ananasban
3	anana$ananasb
4	ananasbanana$
5	anasbanana$an
6	asbanana$anan
7	banana$ananas
8	na$ananasbana
9	nana$ananasba
10	nanasbanana$a
11	nasbanana$ana
12	sbanana$anana

*i*th a in first and last column both correspond to the same a in the text.

Stop and Think: **Does this resolve all ambiguity?**

Inverting BWT

```
 0 $ananasbanana
 1 a$ananasbanan
 2 ana$ananasban
 3 anana$ananasb
 4 ananasbanana$
 5 anasbanana$an
 6 asbanana$anan
 7 banana$ananas
 8 na$ananasbana
 9 nana$ananasba
10 nanasbanana$a
11 nasbanana$ana
12 sbanana$anana
```

- Reconstruct text from its BWT **annb$nnsaaaaa**
- By sorting the BWT, we get the first column!
- Info from each row: first character is preceeded by last.
- Reconstruct from right to left:
  - from line 0, $ is preceeded by a
  - From lines 8–12: a is preceeded by n OR s
  - we need to resolve the ambiguity! DONE
  - $ is preceeded by first a (row 0)
  - first a is preceeded by first n (row 1)
  - first n $\rightarrow$ second a (row 8)
  - second a $\rightarrow$ second n (row 2)
  - ...

Matching backwards in BWT

```
  0 $ananasbanana
→ 1 a$ananasbanan
  2 ana$ananasban
  3 anana$ananasb
  4 ananasbanana$
  5 anasbanana$an
→ 6 asbanana$anan
  7 banana$ananas
  8 na$ananasbana
  9 nana$ananasba
 10 nanasbanana$a
 11 nasbanana$ana
 12 sbanana$anana
```

match **nana**

- (match a) find a in first column (rows 1–6)

Matching backwards in BWT

```
   0 $ananasbanana
   1 a$ananasbanan←
   2 ana$ananasban
   3 anana$ananasb
   4 ananasbanana$
   5 anasbanana$an
   6 asbanana$anan←
   7 banana$ananas
→  8 na$ananasbana
   9 nana$ananasba
  10 nanasbanana$a
→ 11 nasbanana$ana
  12 sbanana$anana
```

match **nana**

- (match a) find a in first column (rows 1–6)
- (match n) preceeded by 1st–4th n (rows 8–11)

Matching backwards in BWT

```
   0 $ananasbanana
   1 a$ananasbanan
→  2 ana$ananasban
   3 anana$ananasb
   4 ananasbanana$
→  5 anasbanana$an
   6 asbanana$anan
   7 banana$ananas
   8 na$ananasbana←
   9 nana$ananasba
  10 nanasbanana$a
  11 nasbanana$ana←
  12 sbanana$anana
```

match **nana**

- (match a) find a in first column (rows 1–6)
- (match n) preceeded by 1st–4th n (rows 8–11)
- (match a) 2nd–5th a (rows 2–5)

Matching backwards in BWT

```
 0 $ananasbanana
 1 a$ananasbanan
 2 ana$ananasban←
 3 anana$ananasb
 4 ananasbanana$
 5 anasbanana$an←
 6 asbanana$anan
 7 banana$ananas
 8 na$ananasbana
→ 9 nana$ananasba
→10 nanasbanana$a
11 nasbanana$ana
12 sbanana$anana
```

match **nana**

- (match a) find a in first column (rows 1–6)
- (match n) preceeded by 1st–4th n (rows 8–11)
- (match a) 2nd–5th a (rows 2–5)
- (match n) 2nd or 3rd n (rows 9,10)

Matching backwards in BWT

```
 0 $ananasbanana
 1 a$ananasbanan
 2 ana$ananasban
 3 anana$ananasb
 4 ananasbanana$
 5 anasbanana$an
 6 asbanana$anan
 7 banana$ananas
 8 na$ananasbana
 9 nana$ananasba
10 nanasbanana$a
11 nasbanana$ana
12 sbanana$anana
```

match **nana**

- (match a) find a in first column (rows 1–6)
- (match n) preceeded by 1st–4th n (rows 8–11)
- (match a) 2nd–5th a (rows 2–5)
- (match n) 2nd or 3rd n (rows 9,10)

$\rightarrow$ 2 occurrences

Matching backwards in BWT

```
 0 $ananasbanana
 1 a$ananasbanan
 2 ana$ananasban
 3 anana$ananasb
 4 ananasbanana$
 5 anasbanana$an
 6 asbanana$anan
 7 banana$ananas
 8 na$ananasbana
 9 nana$ananasba
10 nanasbanana$a
11 nasbanana$ana
12 sbanana$anana
```

match **nana**

- (match a) find a in first column (rows 1–6)
- (match n) preceeded by 1st–4th n (rows 8–11)
- (match a) 2nd–5th a (rows 2–5)
- (match n) 2nd or 3rd n (rows 9,10)

→ 2 occurrences

Stop and Think: How to make it fast?
(mapping last-to-first)

Matching backwards in BWT

```
 0 $ananasbanana
 1 a$ananasbanan
 2 ana$ananasban
 3 anana$ananasb
 4 ananasbanana$
 5 anasbanana$an
 6 asbanana$anan
 7 banana$ananas
 8 na$ananasbana
 9 nana$ananasba
10 nanasbanana$a
11 nasbanana$ana
12 sbanana$anana
```

match **nana**

- (match a) find a in first column (rows 1–6)
- (match n) preceeded by 1st–4th n (rows 8–11)
- (match a) 2nd–5th a (rows 2–5)
- (match n) 2nd or 3rd n (rows 9,10)

→ 2 occurrences

Stop and Think: How to know position of occurrences? (lookup in suffix array)

Optimizing BWT (for mapping)

- **efficient** construction BWT from suffix array (SA)
- for **fast** search: BWT + tables
  - suffix array
  - first occurrence of each character in first column
  - count of each character in columns $0, \ldots, i$
- store tables only **partially**, e.g. for rows 100,200,300,...; recompute in between

Stop and Think: Why is it essential to store tables partially (if we want to improve over suffix arrays)?

Index structures of concrete mapping tools

- **BLAT** (Kent, 2002): k-mer index
- **Segemehl** (Hoffmann. . . , 2009): (Enhanced) suffix array
- **Bowtie** (Langmead. . . , 2009): BWT index
- **Bowtie2** (Langmead. . . , 2012): FM-Index (compressed index based on BWT)

Final thoughts

- "In 1975, the state-of-the-art Aho-Corasick algorithm was touted as requiring 15 minutes to map just **24 English words** to a dictionary. Just a generation later, when Burrows-Wheeler based approaches were introduced into read mapping, the same 15 minutes was used to map almost **ten million reads** to a reference genome." (from Compeau, Pevzner)
- Important algorithmic aspects:
  - time efficiency (construction + matching)
  - space consumption (in construction, index size)
- Mismatch tolerance of mappers?
- Mapping with indels?
- Compare BLAST, Altschul et al., 1990