Strutture dati per indicizzazione testi (genomici)

1

Cosa abbiamo visto

Confronto tra sequenze

- Calcolo della LCS tra 2 sequenze (PD) per studiare la similarità, globale, locale
- Allineamento multiplo

sottosequenza… score…

Sequenziamento

- Shotgun sequencing
- Fragment assembly
  - approccio graph based

2

Cosa abbiamo visto

Confronto tra sequenze

- Calcolo della LCS tra 2 sequenze (PD) per studiare la similarità, globale, locale
- Allineamento multiplo

Sequenziamento

- Shotgun sequencing
- Fragment assembly
  - approccio graph based

sottosequenza… score…

calcolo overlap gestione k-mer (bloom filter)

ma potremmo volere determinare dove una read è maggiormente simile ad un genoma di riferimento

3

2 problemi «nascosti»

calcolo overlap

(BWT e altre strutture dati)

k-mer counting

(conteggio, rappresentazione con Bloom Filter, Hash, minhash, BWT)

4

Argomenti da trattare e «raccontare»

Pan-genome graph
✓ Survey

Strutture di indicizzazione testi
✓ **ST, SA, BWT, LCP [*pattern matching e overlap*]**
- Algoritmo lineare per la costruzione di OG
✓ **BWA e BOWTIE [aligners]** ← ***[seminario-esame]***
✓ ***Lyndon & overlap & sequence embedding***

Tecniche alignment-free
✓ minsketch

5

Milestones

1959: Suffix Tree

1990: Suffix Array (Myers e Manber)

1993: Longest Common Prefix (LCP) Array (Myers e Manber)

1994: Burrow-Wheeler Transform (BWT)

6

How Do We Locate Disease- Causing Mutations?

Combinatorial Pattern Matching

Phillip Compeau and Pavel Pevzner.

Bioinformatics Algorithms: an Active Learning Approach

7

Outline

- **Why do we map reads? [*reference-based*]**
- Using the Trie
- From a Trie to a Suffix Tree
- String Compression and the Burrows-Wheeler Transform
- Inverting Burrows-Wheeler
- Using Burrows-Wheeler for Pattern Matching
- Finding the Matched Patterns
- Setting Up Checkpoints
- Inexact Matching
- Further Applications of Read Mapping

8

Toward a Computational Problem

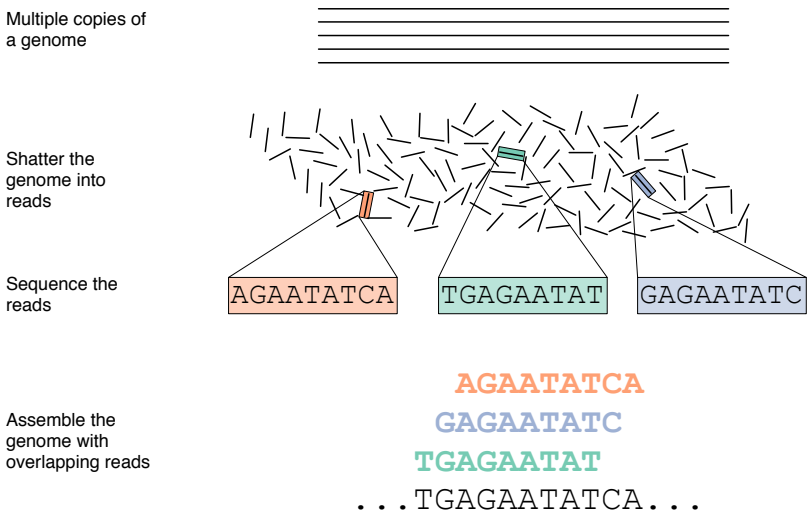
- **Reference genome**: database genome used for comparison.
- **Question:** How can we assemble individual genomes (or represented by reads) efficiently using the reference?

CTGATGATGGACTACGCTACTACTGCTAGCTGTAT Individual

CTGAGGATGGACTACGCTACTACTGATAGCTGTTT *Reference*

9

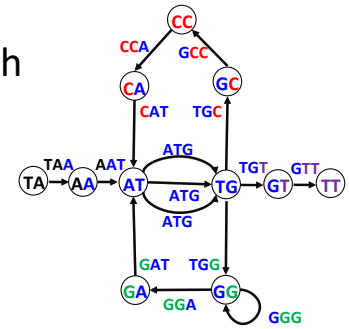
Why Not Use Assembly?



10

Answer1: Why Not Use Assembly?

- Constructing a de Bruijn graph takes a lot of memory.
- Hope: a machine in a clinic that would collect and map reads in 10 minutes.
- Idea: use existing structure of reference genome to help us sequence a patient's genome.



11

Answer2:

https://training.galaxyproject.org/training-material/

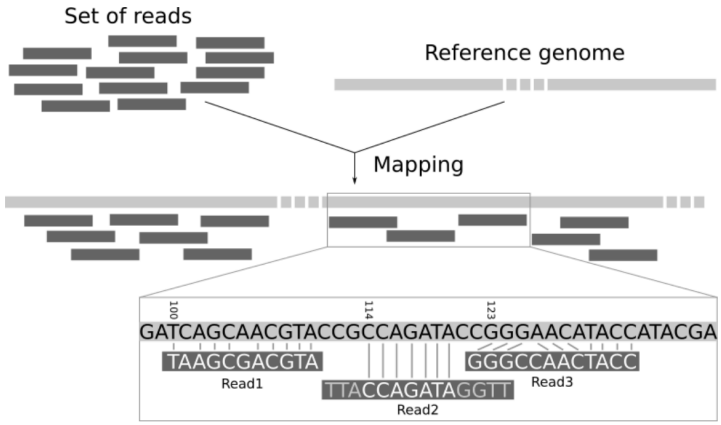
… In principle, we could do a BLAST analysis to figure out where the sequenced pieces fit best in the known genome.

We would need to do that for each of the millions of reads in our sequencing data. Aligning millions of short sequences this way may, however, take a couple of weeks.

And we do not care about the exact base to base correspondence (alignment). What we are interested in is "**where these reads came from**".

This approach is called **mapping**.

12



Open image in new tab

Figure 1: Illustration of the mapping process. The input consists of a set of reads and a reference genome. In the middle, it gives the results of mapping: the locations of the reads on the reference genome. The first read is aligned at position 100 and the alignment has two mismatches. The second read is aligned at position 114. It is a local alignment with clippings on the left and right. The third read is aligned at position 123. It consists of a 2-base insertion and a 1-base deletion.

13

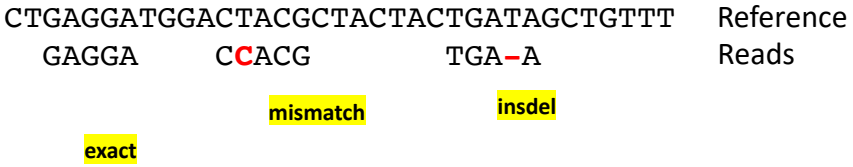
- Input: Set of reads + a reference genome
- Output: BAM file (Binary Alignment Map) file is a compressed binary file storing the read sequences, whether they have been aligned to a reference sequence (e.g. a chromosome), and if so, the position on the reference sequence at which they have been aligned

14

Read Mapping

- **Read mapping**: determine where each read has high similarity to the reference genome.

without assemblying



15

Exact Pattern Matching

- Focus on a simple question: where do the reads match the reference genome *exactly*?

- **Single Pattern Matching Problem:**
  - **Input:** A string *Pattern* (read) and a string *Genome.*
  - **Output:** All positions in *Genome* where *Pattern* appears as a substring.

16

Exact Pattern Matching

- Focus on a simple question: where do the reads match the reference genome *exactly*?
- **Multiple Pattern Matching Problem:**
  - **Input:** A collection of strings *Patterns* (reads) and a string *Genome.*
  - **Output:** All positions in *Genome* where a string from *Patterns* appears as a substring.

17

A Brute Force Approach

- We can simply iterate a brute force approach method, *sliding* each *Pattern* down *Genome*.

```
panamabananas   Genome
Pattern
```

- **Note**: we use words instead of DNA strings for convenience.

18

A Brute Force Approach

- We can simply iterate a brute force approach method, sliding each *Pattern* down *Genome*.

```
panamabananas   Genome
nana            Pattern
```

- **Note**: we use words instead of DNA strings for convenience.

19

A Brute Force Approach

- We can simply iterate a brute force approach method, sliding each *Pattern* down *Genome*.

```
panamabananas   Genome
nana            Pattern
```

- **Note**: we use words instead of DNA strings for convenience.

20

A Brute Force Approach

- We can simply iterate a brute force approach method, sliding each *Pattern* down *Genome*.

```
p a n a m a b a n a n a s      Genome
n a n a                        Pattern
```

- **Note**: we use words instead of DNA strings for convenience.

21

A Brute Force Approach

- We can simply iterate a brute force approach method, sliding each *Pattern* down *Genome*.

```
p a n a m a b a n a n a s      Genome
  n a n a                      Pattern
```

- **Note**: we use words instead of DNA strings for convenience.

22

A Brute Force Approach

- We can simply iterate a brute force approach method, sliding each *Pattern* down *Genome*.

```
p a n a m a b a n a n a s      Genome
  n a n a                      Pattern
```

- **Note**: we use words instead of DNA strings for convenience.

23

A Brute Force Approach

- We can simply iterate a brute force approach method, sliding each *Pattern* down *Genome*.

```
p a n a m a b a n a n a s      Genome
  n a n a                      Pattern
```

- **Note**: we use words instead of DNA strings for convenience.

24

A Brute Force Approach

- We can simply iterate a brute force approach method, sliding each *Pattern* down *Genome*.

```
p a n a m a b a n a n a s     Genome
n a n a                       Pattern
```

- **Note**: we use words instead of DNA strings for convenience.

25

A Brute Force Approach

- We can simply iterate a brute force approach method, sliding each *Pattern* down *Genome*.

```
p a n a m a b a n a n a s     Genome
  n a n a                     Pattern
```

- **Note**: we use words instead of DNA strings for convenience.

26

A Brute Force Approach

- We can simply iterate a brute force approach method, sliding each *Pattern* down *Genome*.

```
panamabananas   Genome
n ana           Pattern
```

- **Note**: we use words instead of DNA strings for convenience.

27

A Brute Force Approach

- We can simply iterate a brute force approach method, sliding each *Pattern* down *Genome*.

```
panamabananas   Genome
 n ana          Pattern
```

- **Note**: we use words instead of DNA strings for convenience.

28

A Brute Force Approach

- We can simply iterate a brute force approach method, sliding each *Pattern* down *Genome*.

```
p a n a m a b a n a n a s      Genome
      n a n a                  Pattern
```

- **Note**: we use words instead of DNA strings for convenience.

29

A Brute Force Approach

- We can simply iterate a brute force approach method, sliding each *Pattern* down *Genome*.

```
p a n a m a b a n a n a s      Genome
      n a n a                  Pattern
```

- **Note**: we use words instead of DNA strings for convenience.

30

A Brute Force Approach

- We can simply iterate a brute force approach method, sliding each *Pattern* down *Genome.*

```
p a n a m a b a n a n a s     Genome
      n a n a                 Pattern
```

- **Note**: we use words instead of DNA strings for convenience.

31

A Brute Force Approach

- We can simply iterate a brute force approach method, sliding each *Pattern* down *Genome.*

```
p a n a m a b a n a n a s     Genome
      n a n a                 Pattern
```

- **Note**: we use words instead of DNA strings for convenience.

32

A Brute Force Approach

- We can simply iterate a brute force approach method, sliding each *Pattern* down *Genome.*

```
p a n a m a b a n a n a s      Genome
      n a n a                  Pattern
```

- **Note**: we use words instead of DNA strings for convenience.

33

A Brute Force Approach

- We can simply iterate a brute force approach method, sliding each *Pattern* down *Genome*.

```
p a n a m a b a n a n a s      Genome
            n a n a            Pattern
```

- **Note**: we use words instead of DNA strings for convenience.

34

That Was Easy!

35

Brute Force Is Too Slow

- The runtime of the brute force approach is too high!
  - Single *Pattern*: O(|*Genome*| * |*Pattern*|)
  - Multiple *Patterns*: O(|*Genome*| * |*Patterns*|)
  - |*Patterns*| = combined length of *Patterns*

36

Outline

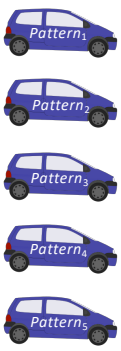
- Why do we map reads?
- **Using the Trie**
- From a Trie to a Suffix Tree
- String Compression and the Burrows-Wheeler Transform
- Inverting Burrows-Wheeler
- Using Burrows-Wheeler for Pattern Matching
- Finding the Matched Patterns
- Setting Up Checkpoints
- Inexact Matching
- Further Applications of Read Mapping

37

Patterns Travel One at a Time

Genome

CTGATGATGGACTACGCTACTACTGCTAGCTGTATTACGATCAGCTAC**C**ACATCGTAGCTAC



38

Packing Patterns onto a Bus

Genome

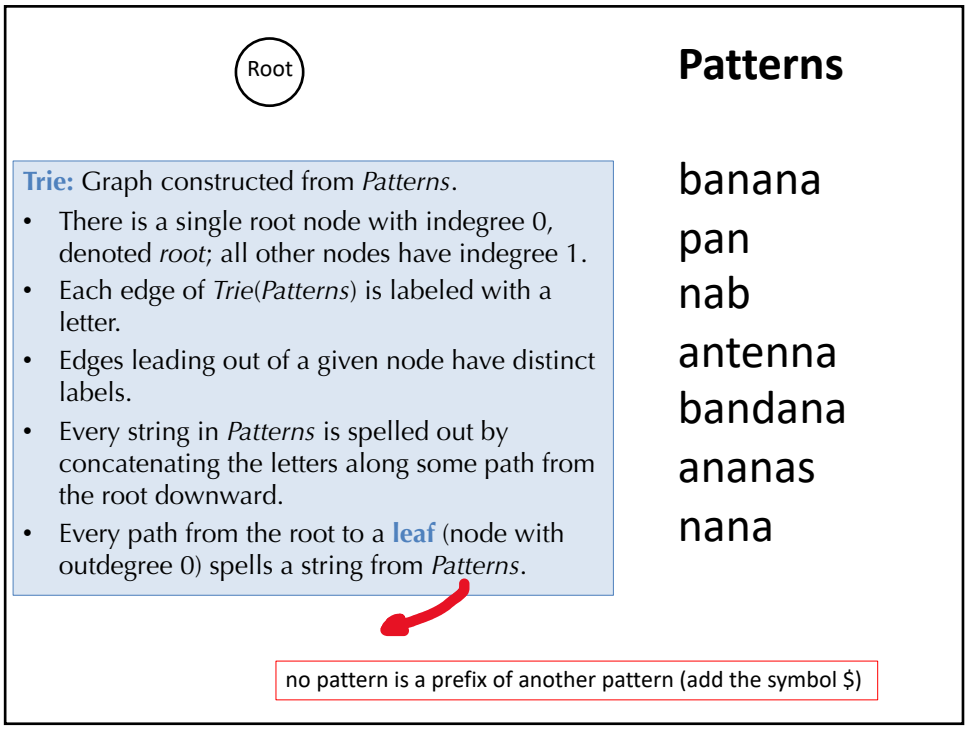
CTGATGATGGACTACGCTACTACTGCTAGCTGTATTACGATCAGCTAC**C**ACATCGTAGCTAC

39

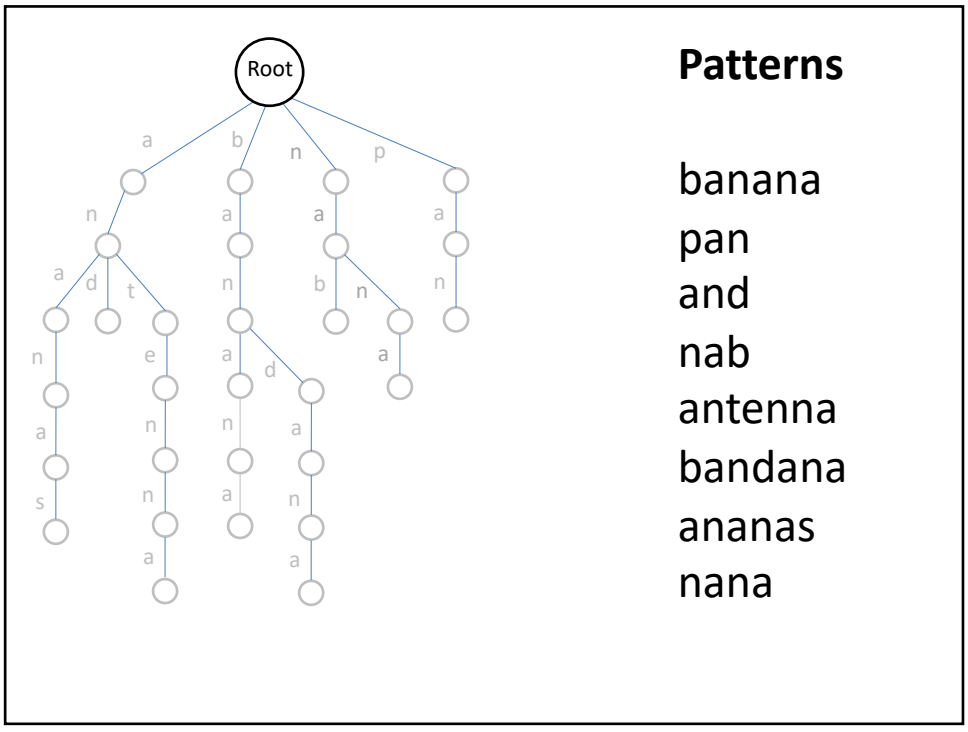
Processing Patterns into a Trie

- Idea: combine reads into a graph. *Each substring of the genome can match at most one read (no read is a prefix of another read).* So each read will correspond to a unique path through this graph.
- The resulting graph [represents a collection of strings] is called a **trie**.

40


Root
Patterns
banana
pan
nab
antenna
bandana
ananas
nana
Trie: Graph constructed from Patterns.
• There is a single root node with indegree 0, denoted root; all other nodes have indegree 1.
• Each edge of Trie(Patterns) is labeled with a letter.
• Edges leading out of a given node have distinct labels.
• Every string in Patterns is spelled out by concatenating the letters along some path from the root downward.
• Every path from the root to a leaf (node with outdegree 0) spells a string from Patterns.
no pattern is a prefix of another pattern (add the symbol $)

41


Root
Patterns
banana
pan
and
nab
antenna
bandana
ananas
nana

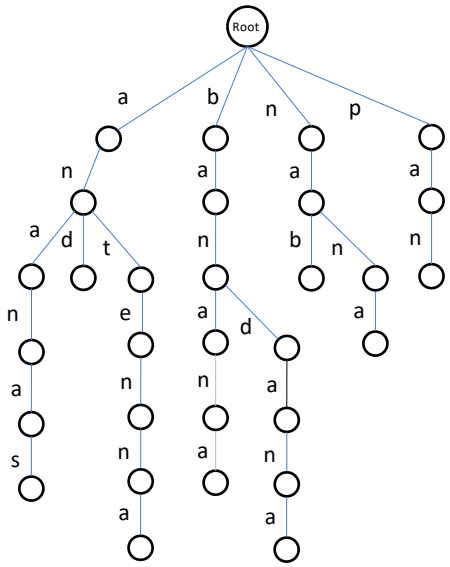
42

Using the Trie for Pattern Matching

- **TrieMatching**: *Slide* the trie down the genome.
- At each position, walk down the trie and see if we can reach a leaf by matching symbols.
- Analogy: bus stops

43

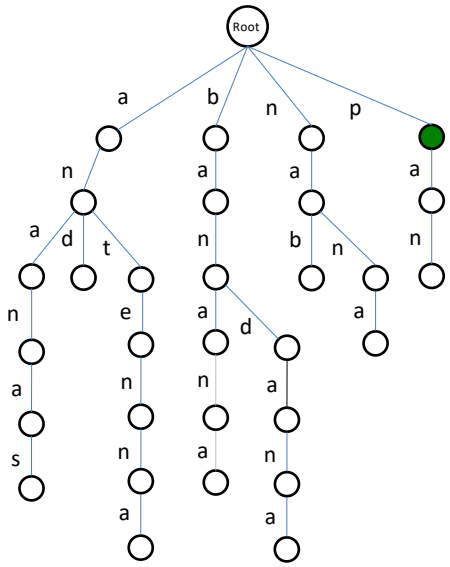
p a n a m a b a n a n a s



Given the genome and the trie constructed from the set of pattern we want to locate

44

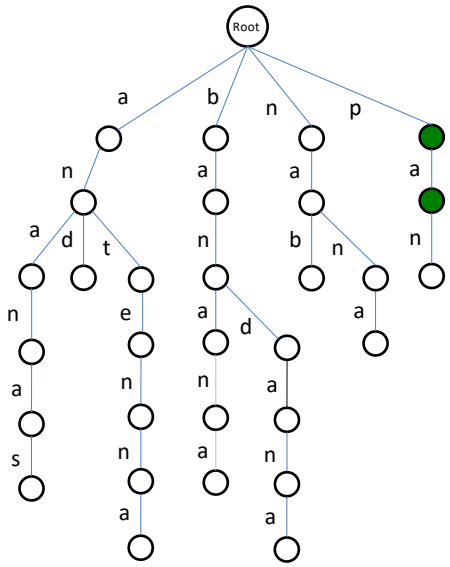
p a n a m a b a n a n a s



Is there a pattern in this collection that can match at this position in the genome?

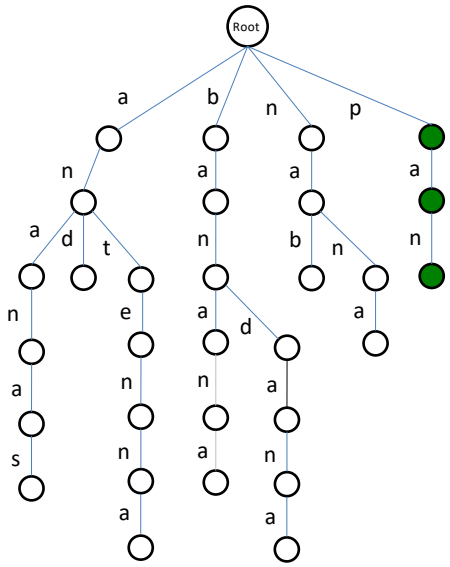
45

p a n a m a b a n a n a s



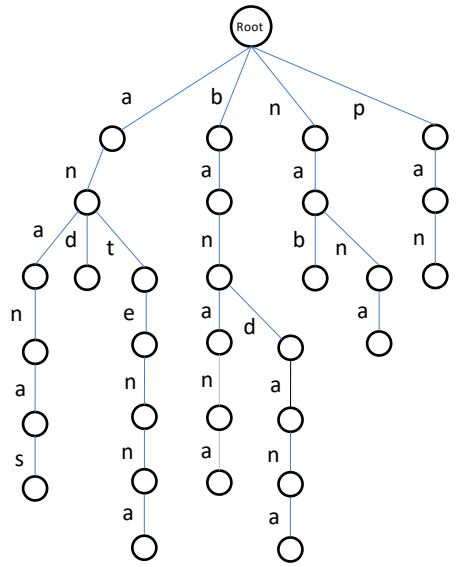
46

p a n a m a b a n a n a s



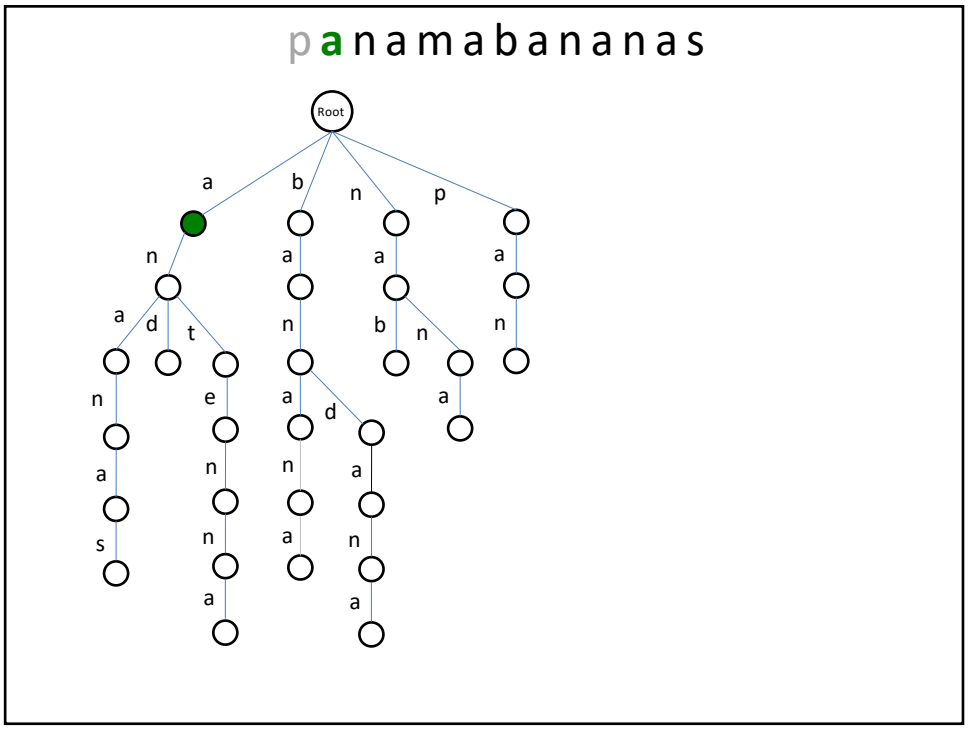
47

p a n a m a b a n a n a s

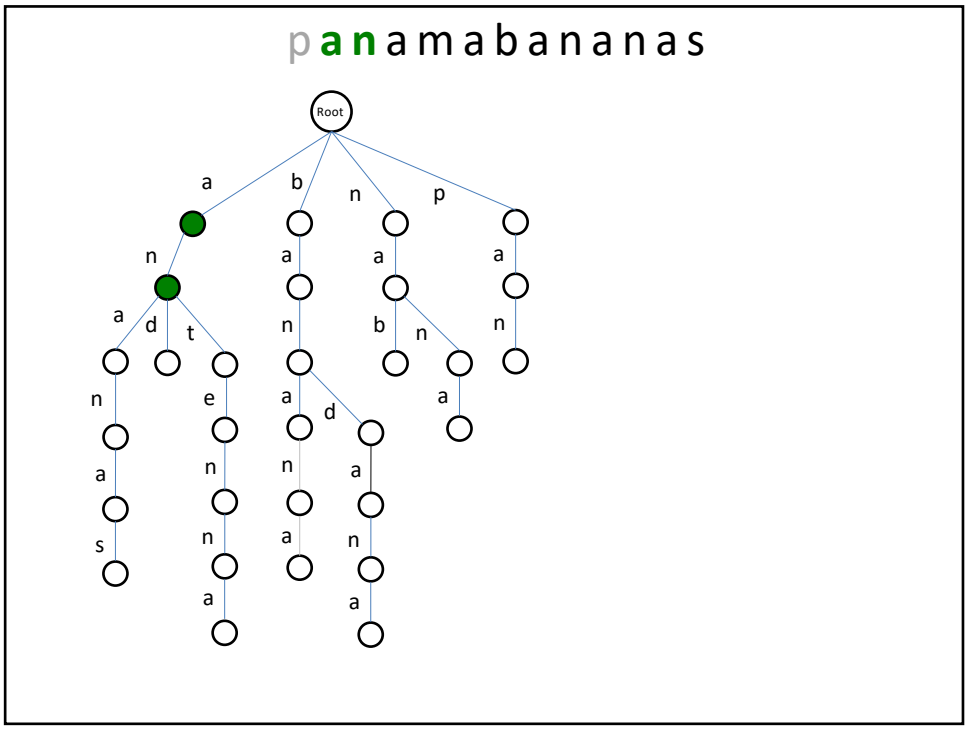


We move the trie and we ask:
Is there a pattern in this collection that can match at this position in the genome?

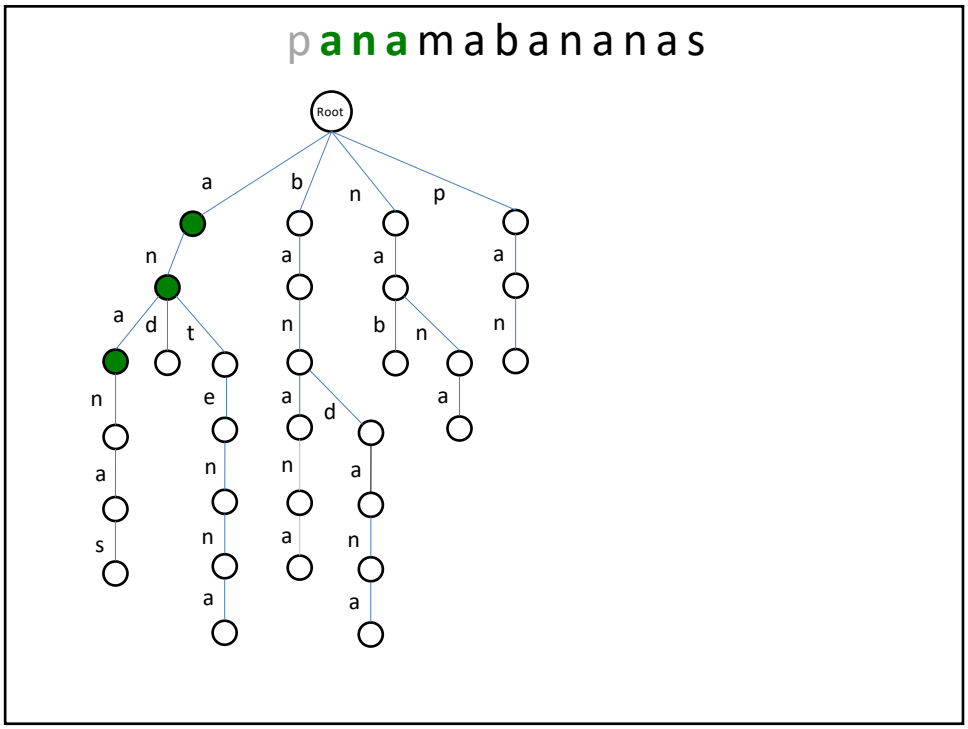
48


p a n a m a b a n a n a s
Root
a
b
n
p
n
a
a
a
a
d
t
n
b
n
n
n
e
a
d
a
n
n
a
n
a
s
n
a
a
n
a
a

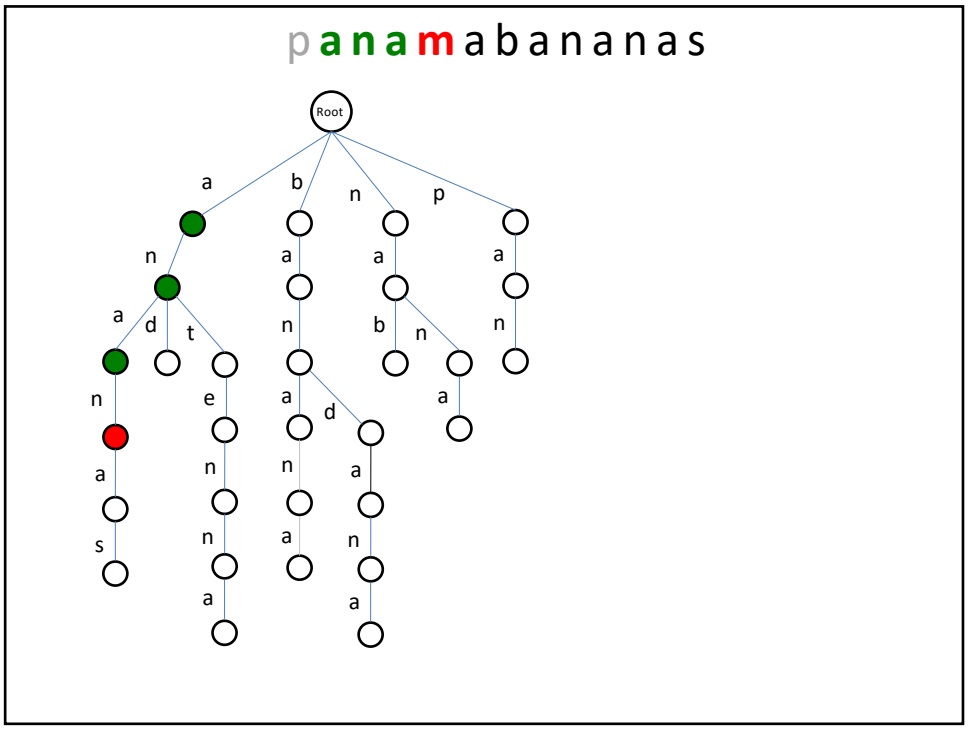
49


p a n a m a b a n a n a s
Root
a
b
n
p
n
a
a
a
a
d
t
n
b
n
n
n
e
a
d
a
n
n
a
n
a
s
n
a
a
n
a
a

50

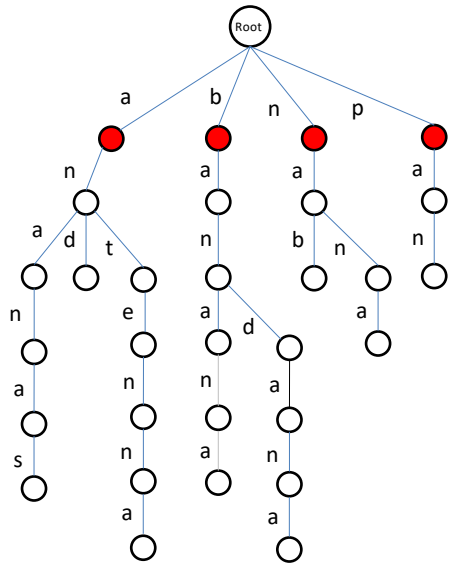

p a n a m a b a n a n a s
Root
a
b
n
p
n
a
a
a
a
d
t
n
b
n
n
n
e
a
d
a
n
n
n
a
n
a
a
s
n
a
a
n
a

51


p a n a m a b a n a n a s
Root
a
b
n
p
n
a
a
a
a
d
t
n
b
n
n
n
e
a
d
a
n
n
n
a
n
a
a
s
n
a
a
n
a

52

p a n a m a b a n a n a **s**



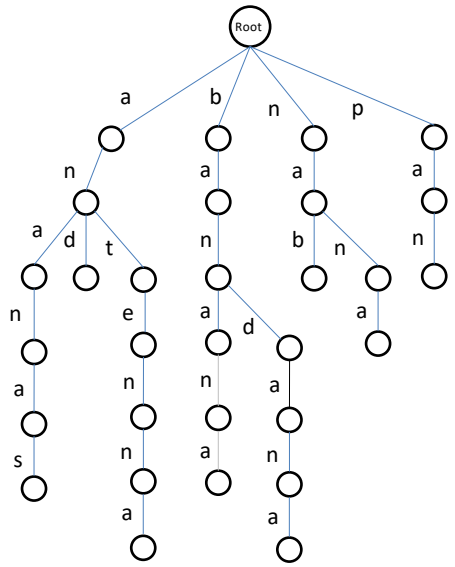
53

p a n a m a b a n a n a **s**

STOP: If we add the pattern "band" to our dataset, why would it cause problems?

Answer: "band" is a *prefix* of "bandana" and would not end at a leaf node …

Adding an "End" Symbol Fixes the Issue



54

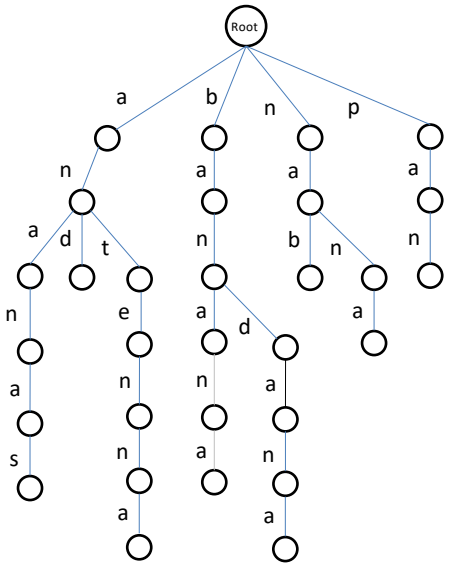
Success!

- Runtime of Brute Force (before):
  - Total: O(|*Genome*|*|*Patterns*|)
- Runtime of Trie Matching:
  - Trie Construction: O(|*Patterns*|)
  - Pattern Matching: O(|*Genome*| * |*LongestPattern*|)

55

Memory Analysis of TrieMatching

- We completely forgot about memory!
- Our trie: 30 edges, |*Patterns*| = 39
- Worst case: # edges = O(|*Patterns*|)

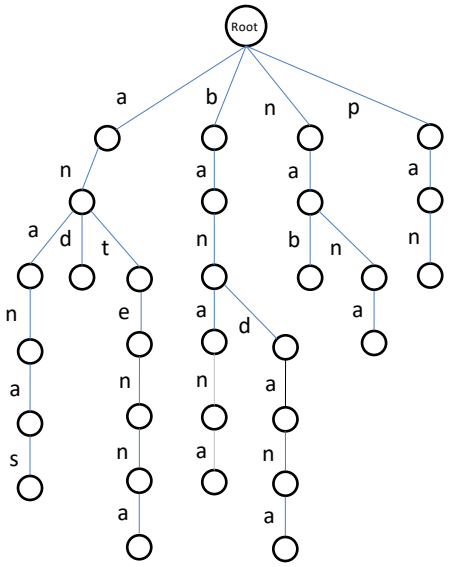


56

Memory Analysis of TrieMatching

- We completely forgot about memory!
- Our trie: 30 edges, |*Patterns*| = 39

Worst case: Trie takes up $O(|Patterns|)$, which for a human sequencing project may be ~1 TB!



57

Outline

- Why do we map reads?
- Using the Trie
- **From a Trie to a Suffix Tree**
- String Compression and the Burrows-Wheeler Transform
- Inverting Burrows-Wheeler
- Using Burrows-Wheeler for Pattern Matching
- Finding the Matched Patterns
- Setting Up Checkpoints
- Inexact Matching
- Further Applications of Read Mapping

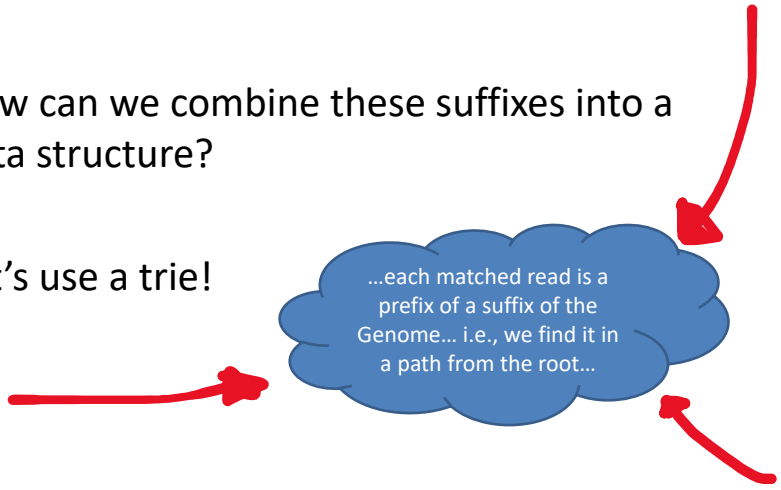
58

PREPROCESSING THE GENOME INSTEAD

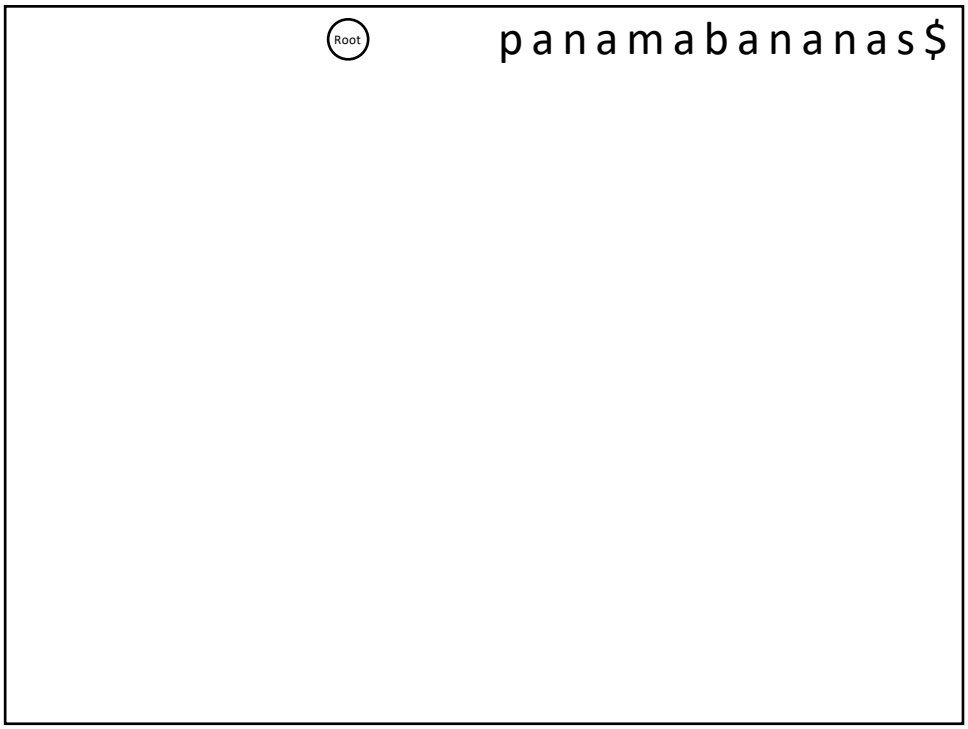
59

Preprocessing the Genome

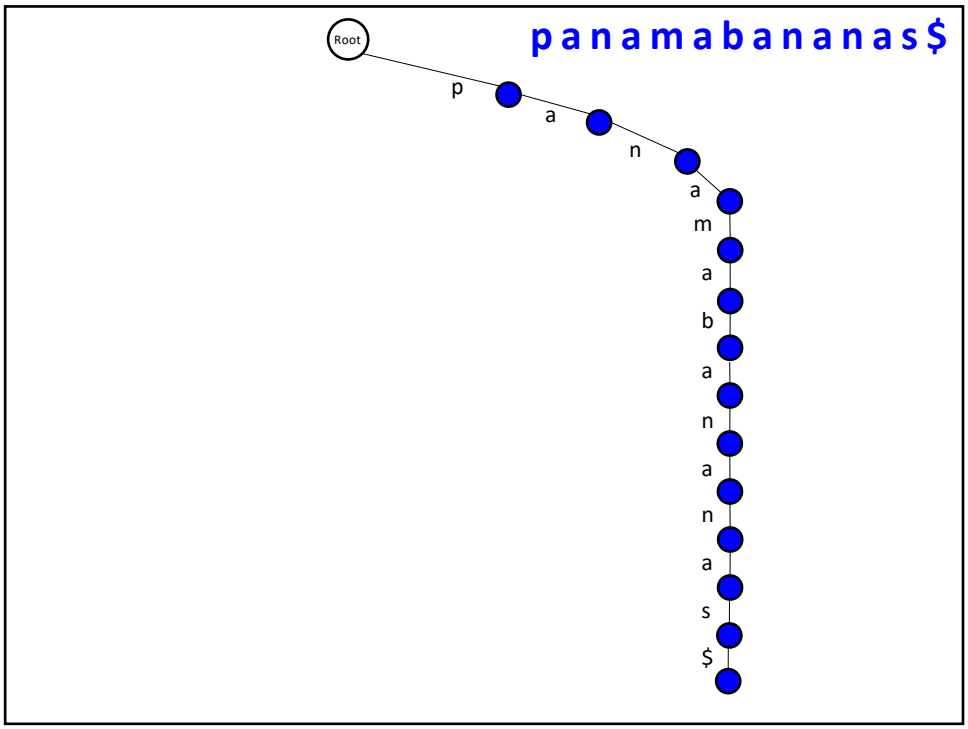
- Split *Genome* into all its suffixes.
- How can we combine these suffixes into a data structure?
- Let's use a trie!



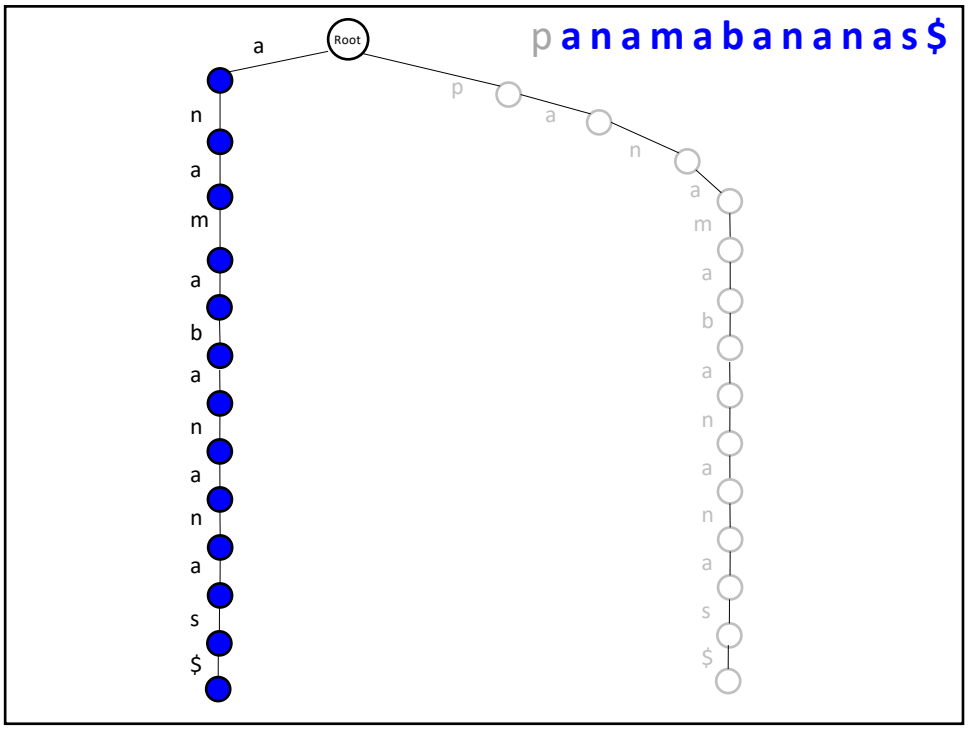
60



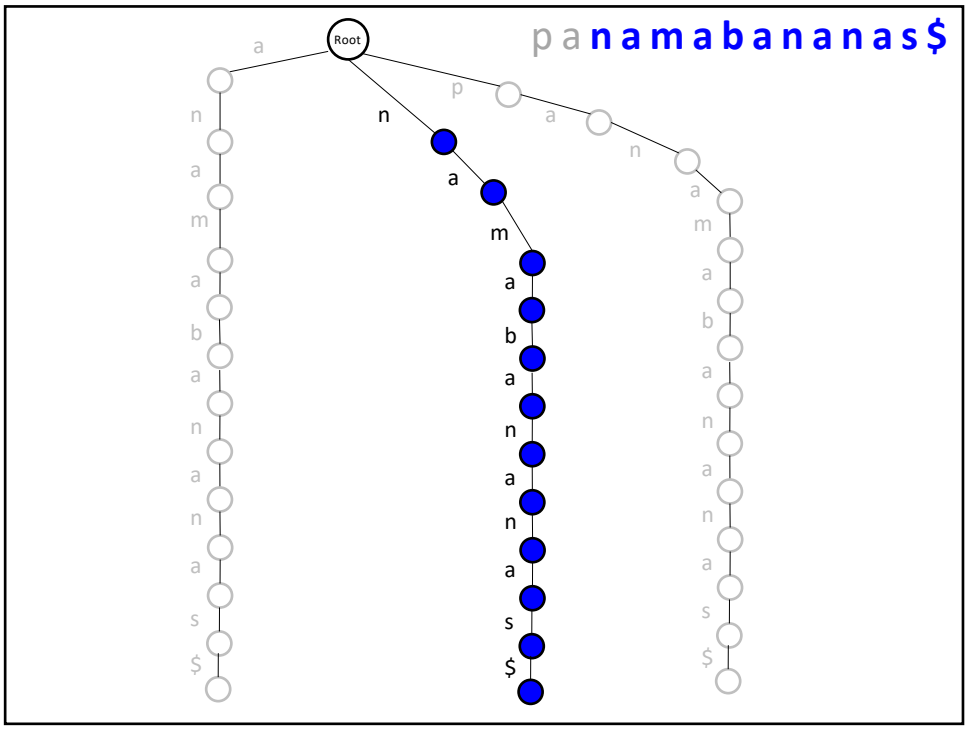
61



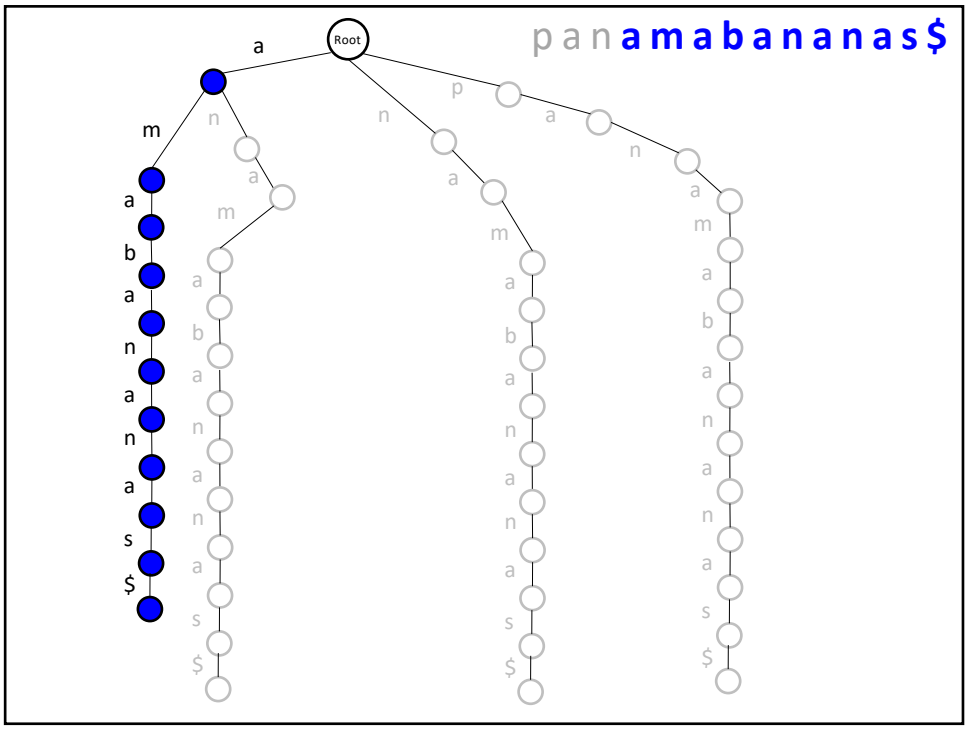
62


panamabananas$
Root
a
n
a
m
a
b
a
n
a
n
a
s
$
p
a
n
a
m
a
b
a
n
a
n
a
s
$

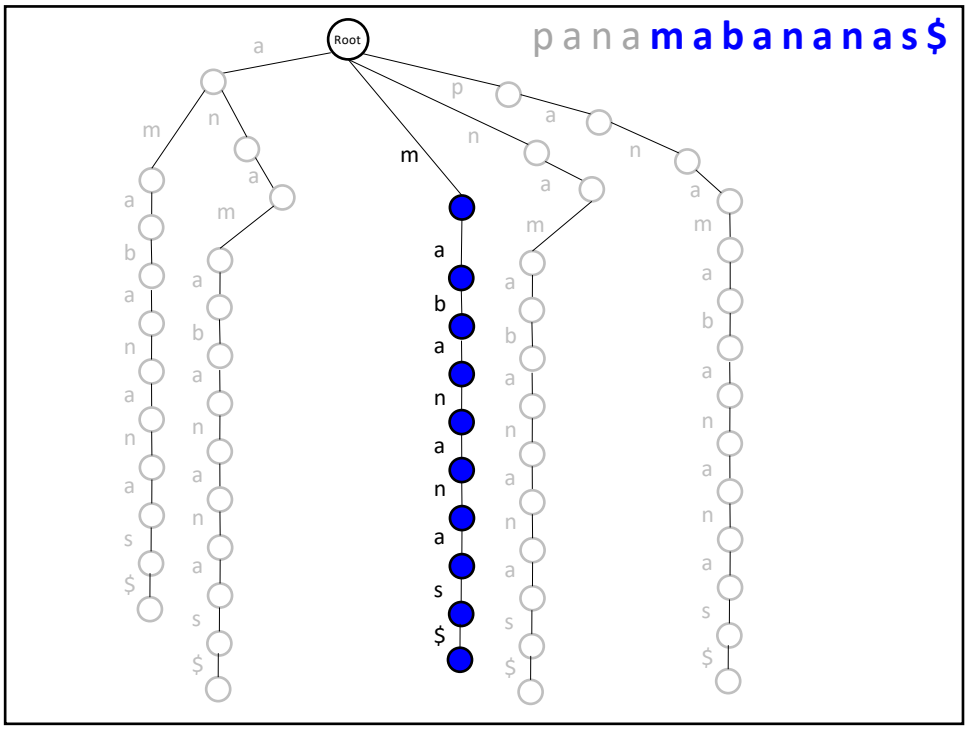
63


panamabananas$
Root
a
n
a
m
a
b
a
n
a
n
a
s
$
p
a
n
a
m
a
b
a
n
a
n
a
s
$

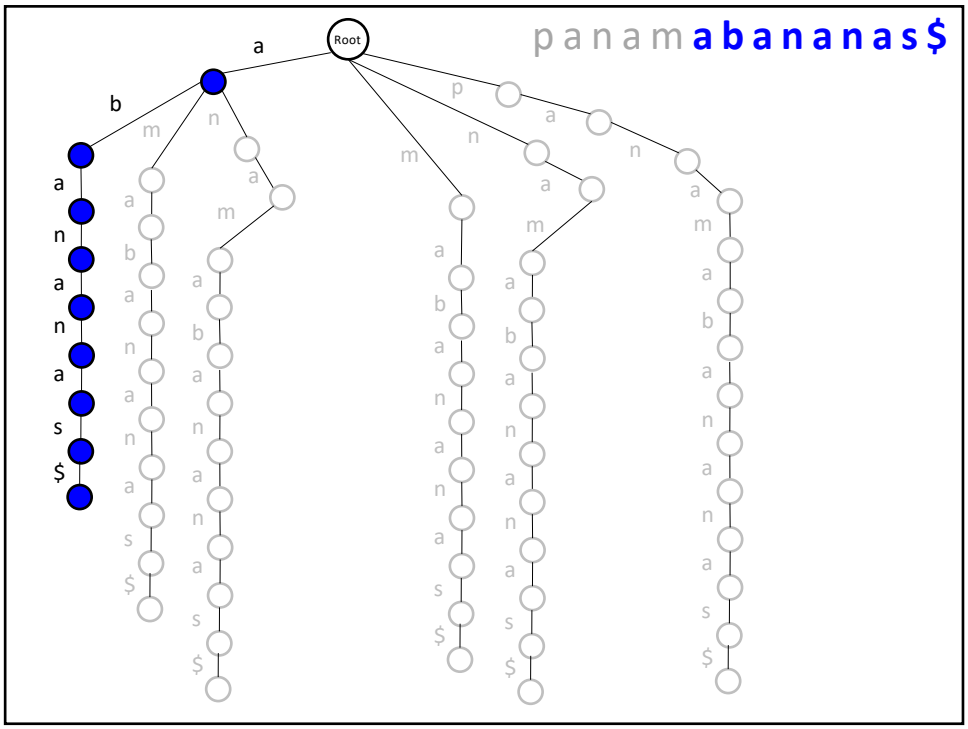
64


panamabananas$
Root
a
m
a
b
a
n
a
n
a
s
$

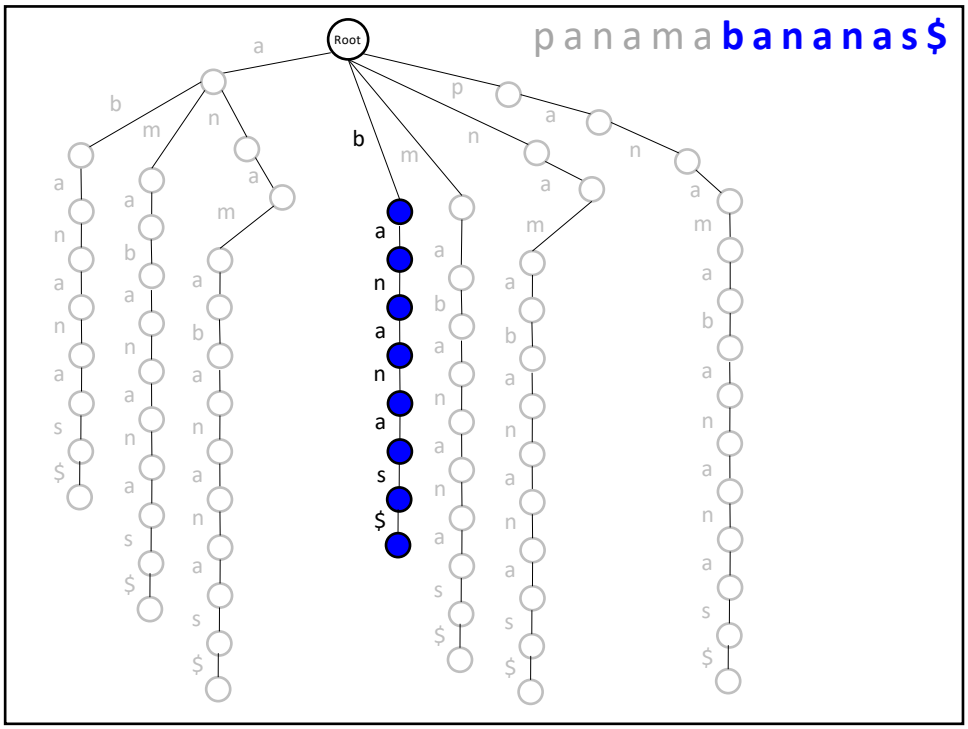
65


panamabananas$
Root
m
a
b
a
n
a
n
a
s
$

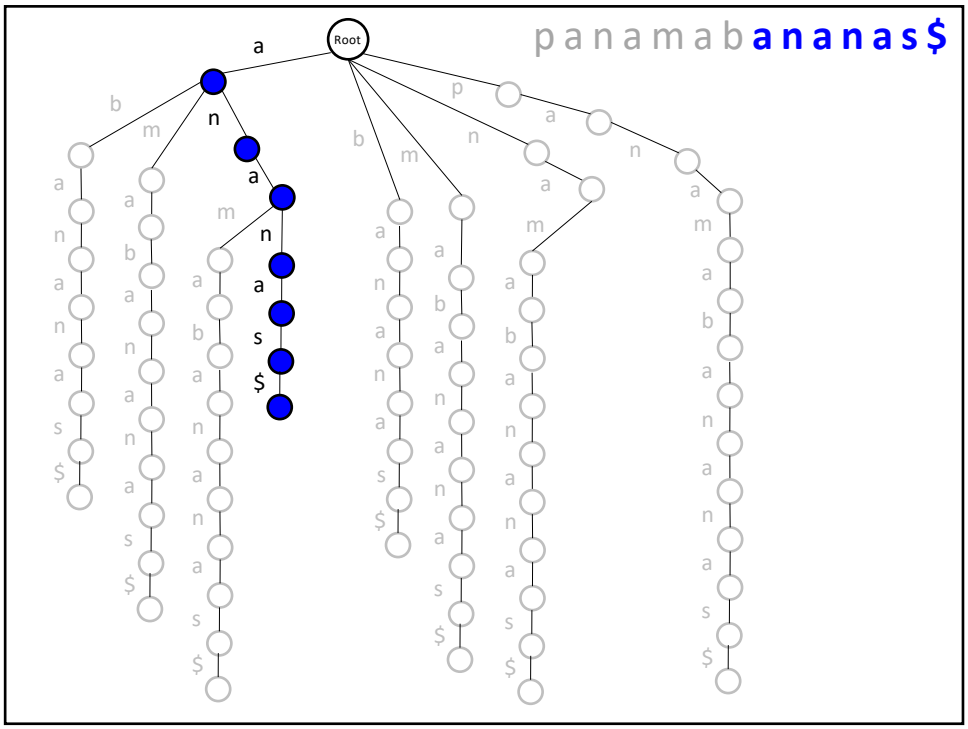
66



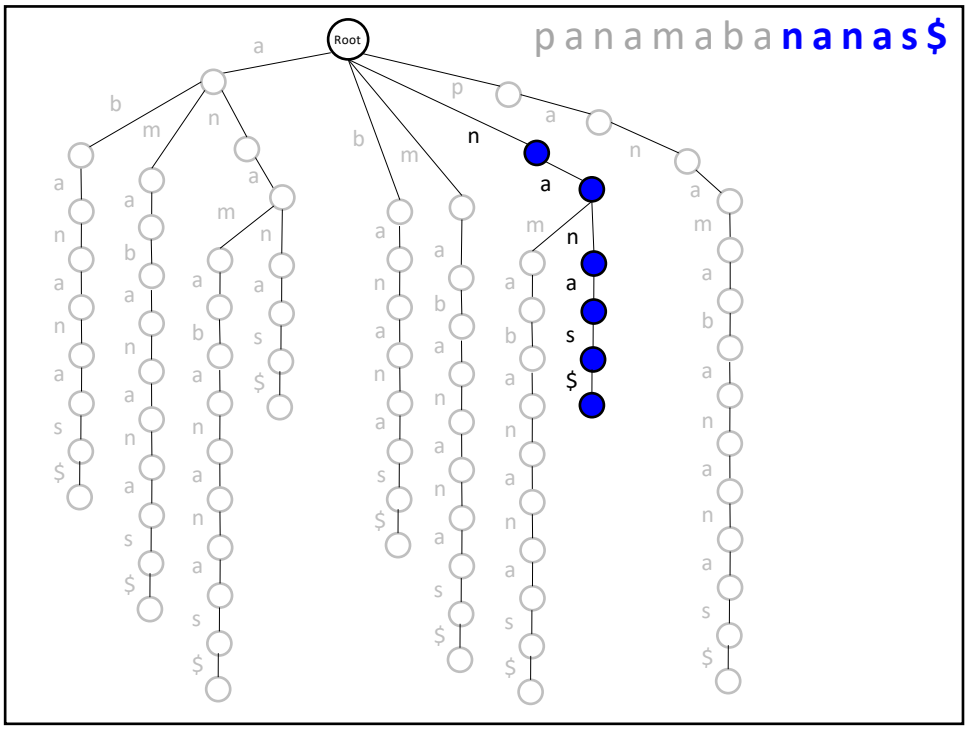
67



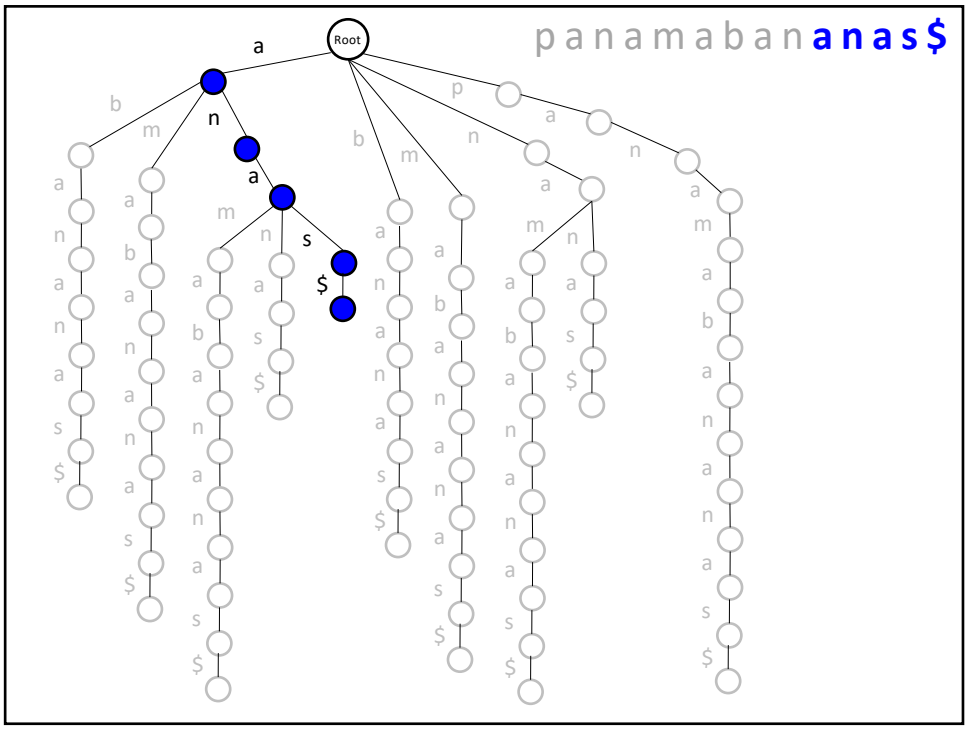
68



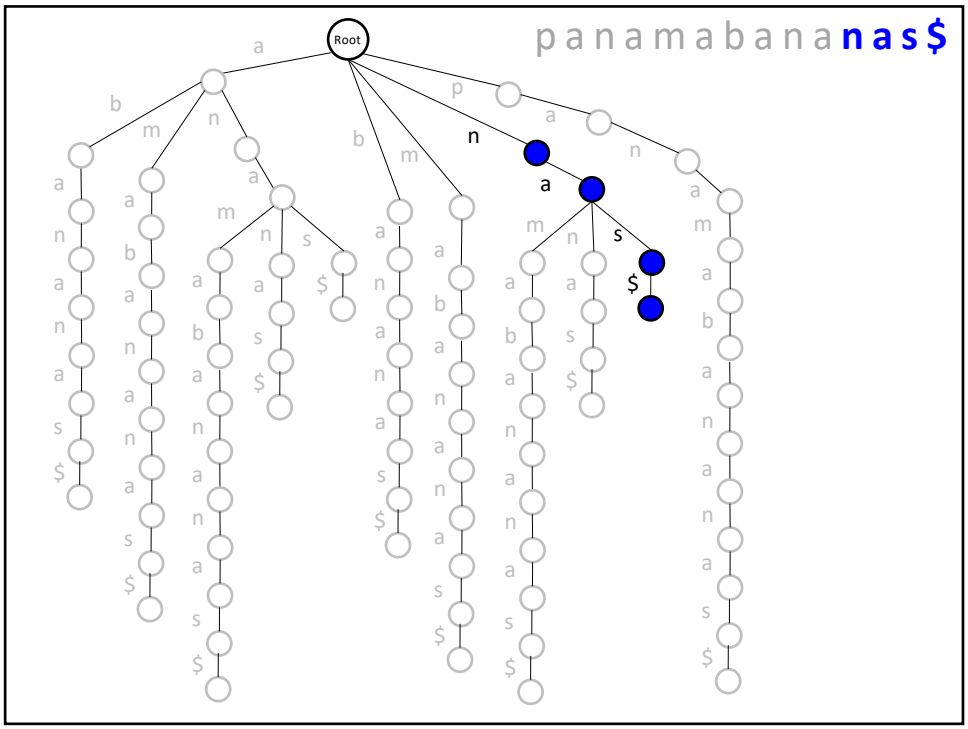
69



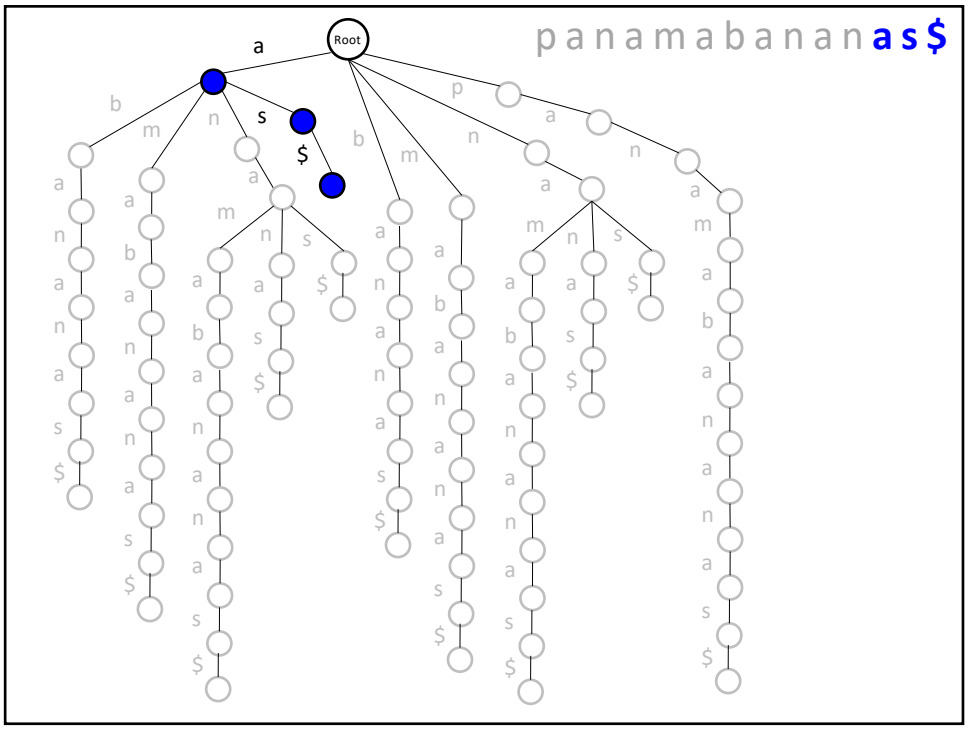
70



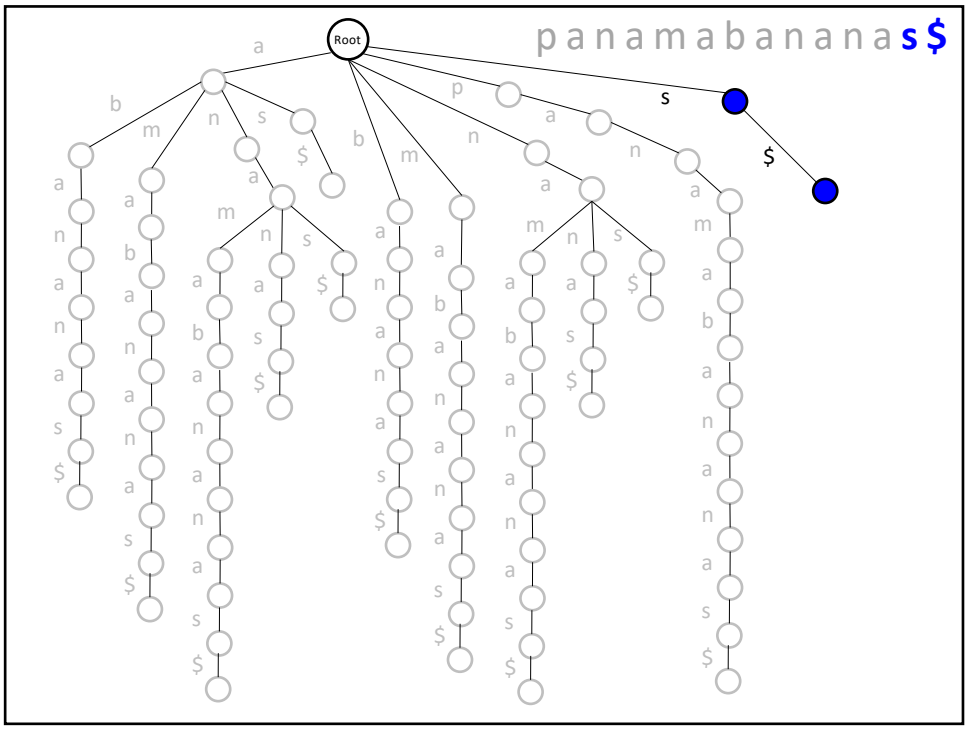
71



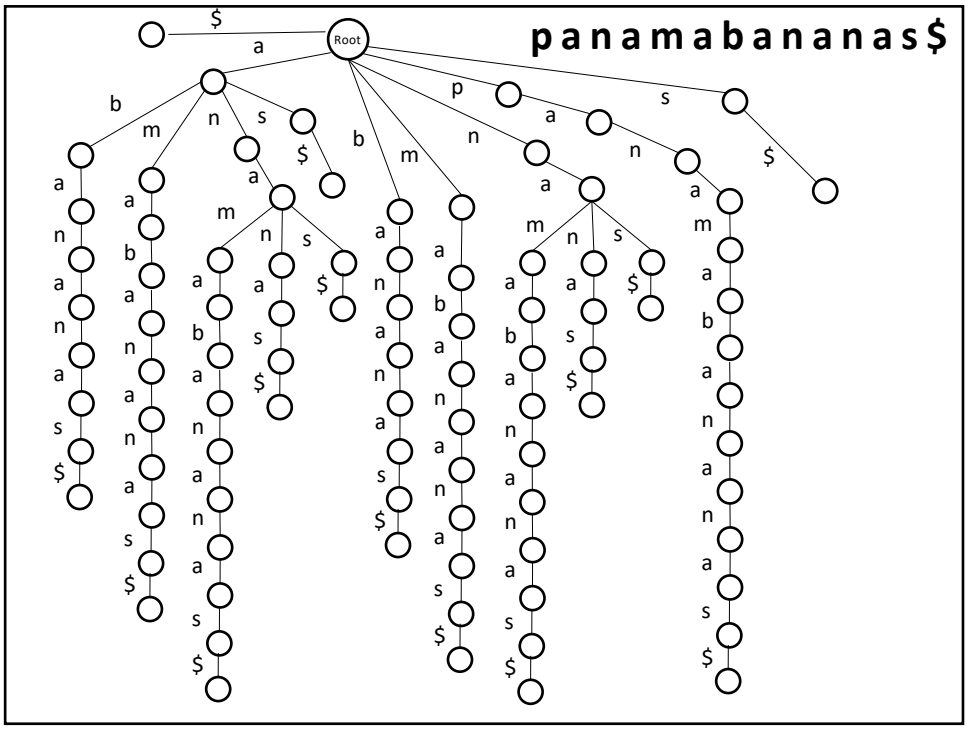
72



73



74



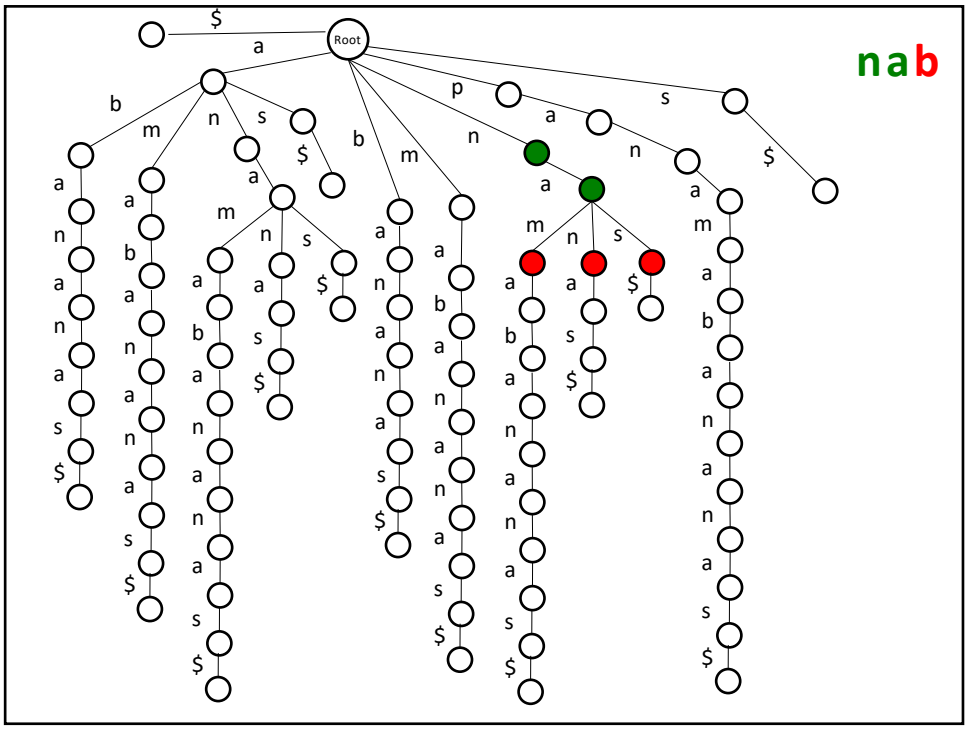
75

The Suffix Trie and Pattern Matching

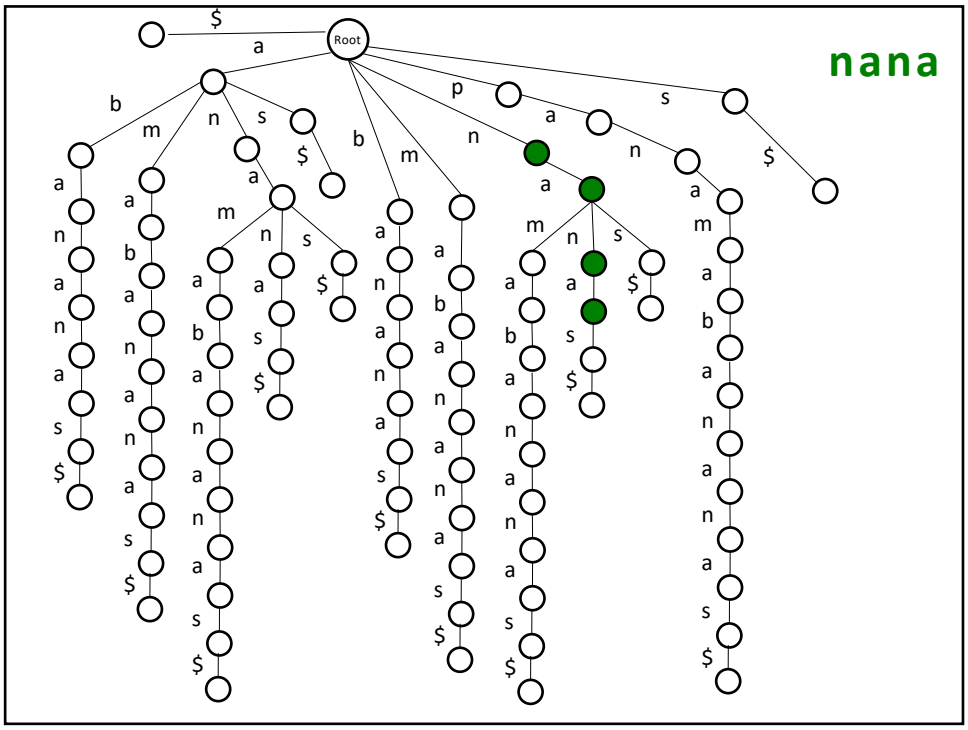
- For each *Pattern*, see if *Pattern* can be spelled out from the root downward in the suffix trie.

Suffix automaton

76



77

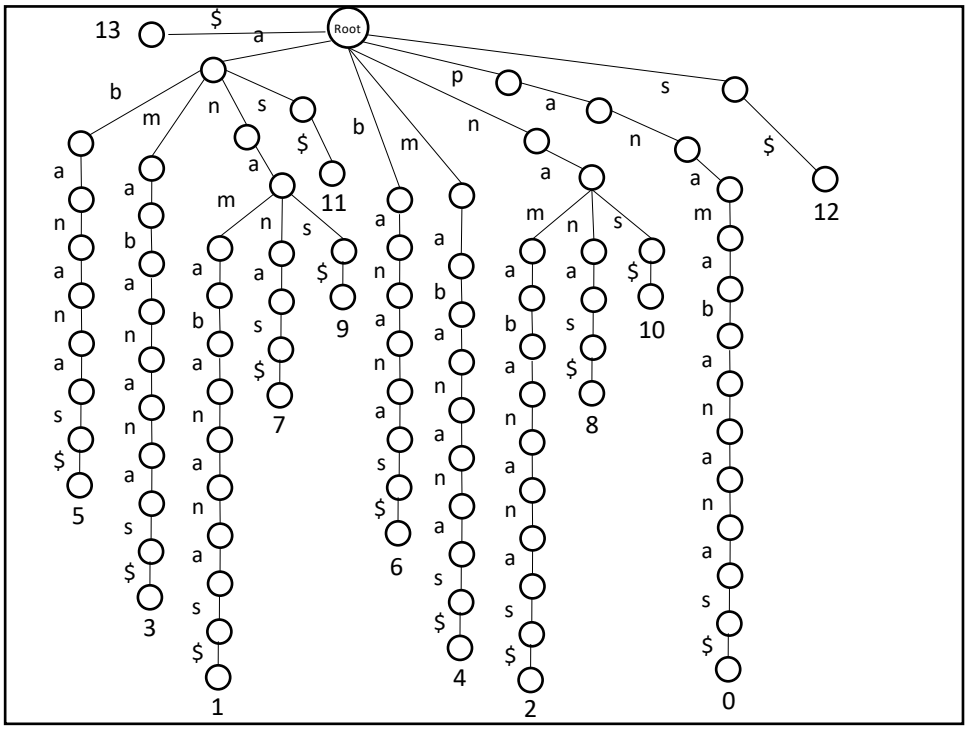


78

Where Are the Matches?

- To find where the pattern matches are, we need to add a little more information to the suffix trie.
- At each leaf ($), we add the starting position in *Genome* of the suffix ending at that leaf.

79

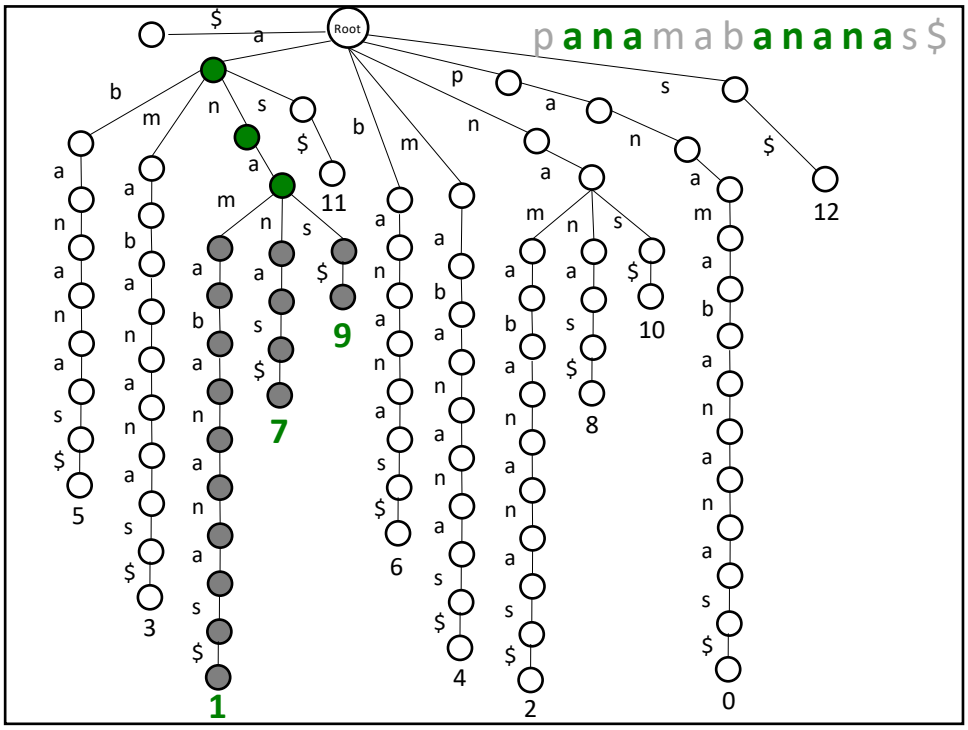


80

Where Are the Matches?

- Once we find a match, we “walk down” to the leaf in order to find the starting position of the match.

81



82

Memory Trouble Once Again

- Worst case: the suffix trie holds O(|*Suffixes*|) nodes.
- For a *Genome* of length *n*, |*Suffixes*| = $n(n-1)/2 = O(n^2)$

Suffixes

panamabananas$
anamabananas$
namabananas$
amabananas$
mabananas$
abananas$
bananas$
ananas$
nanas$
anas$
nas$
as$
s$
$

83

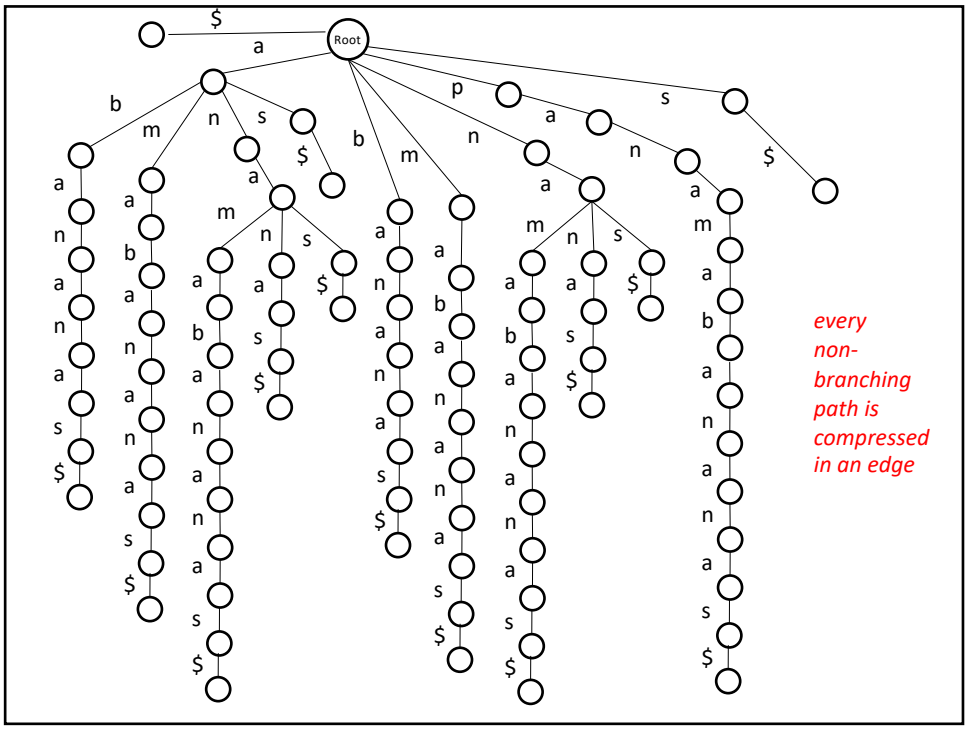
Compressing the Trie

- This doesn't mean that our idea was bad!
- To reduce memory, we can compress each "nonbranching path" of the tree into an edge.

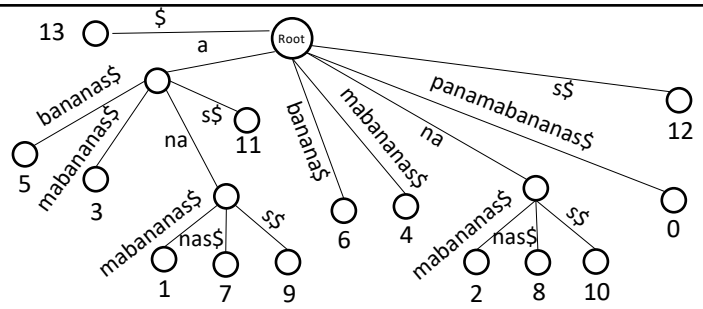
84

SUFFIX TREES

85



86



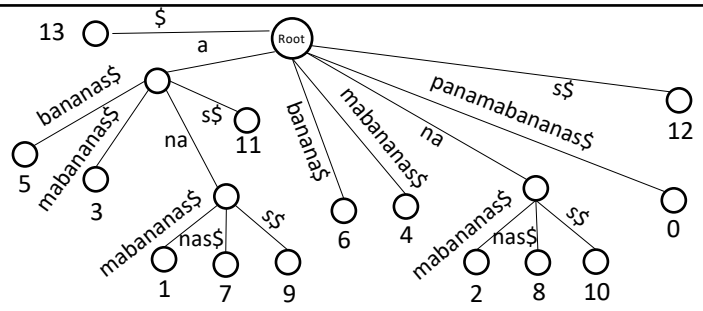
- This data structure is called a **suffix tree**.

Linear pattern matching algorithms - IEEE Conference ...

by P Weiner · 1973 · Cited by 2270 · Related articles

A **linear** time **algorithm** for obtaining a compacted version of a bi-tree associated with a given **string** is presented. ... With this construction as the basic tool, we indicate how to solve several **pattern matching** problems, including some from [4] in **linear** time.

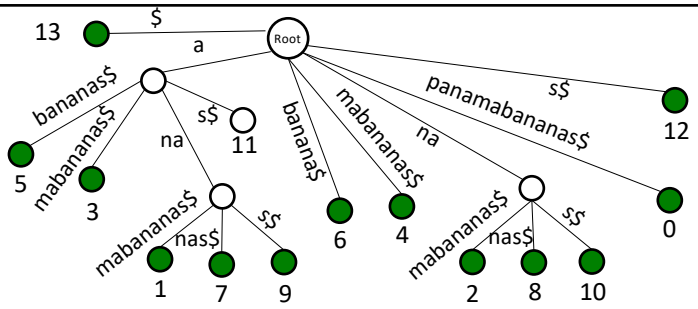
87



- This data structure is called a **suffix tree**.
- For any *Genome*, # nodes < 2|*Genome*| = O(|*Genome*|).

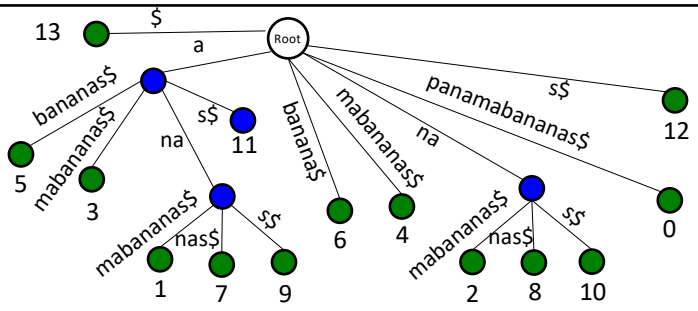
WHY?

88



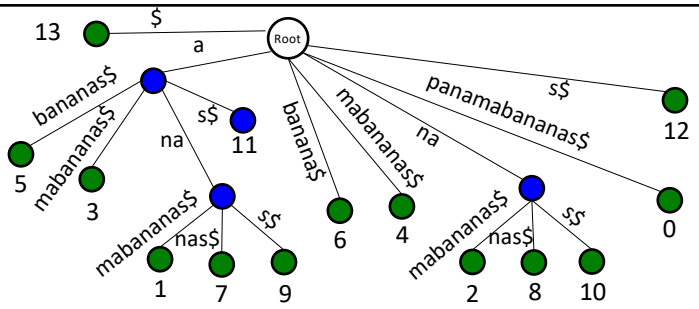
- This data structure is called a **suffix tree**.
- For any *Genome*, # nodes < 2|*Genome*|.
  - **# leaves = |*Genome*|**;

89



- This data structure is called a **suffix tree**.
- For any *Genome*, # nodes < 2|*Genome*|.
  - **# leaves** = |*Genome*|;
  - **# internal nodes** < |*Genome*| – 1 [not easy proof]

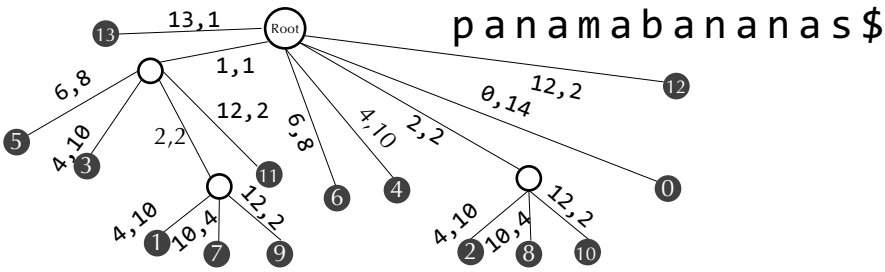
90



STOP: But wait … we are still storing all the suffixes! So why are suffix trees more memory efficient than suffix tries?

Answer: We can replace each edge with two integers: the starting position of the string, and its length.

91



STOP: But wait … we are still storing all the suffixes! So why are suffix trees more memory efficient than suffix tries?

Answer: We can replace each edge with two integers: the starting position of the string, and its length.

92

Runtime and Memory Analysis

- Runtime:
  - $O(|Genome|^2)$ to construct the suffix tree.
  - $O(|Genome| + |Patterns|)$ to find pattern matches.
- Memory:
  - $O(|Genome|^2)$ to construct the suffix tree.
  - $O(|Genome|)$ to store the suffix tree.

93

Runtime and Memory Analysis

- Runtime:
  - $O(|Genome|)$ to construct the suffix tree ***directly.***
- Memory:
  - $O(|Genome|)$ to construct the suffix tree ***directly.***

94

Time complexity

- ✓ Weiner's algorithm (1973)
  - ❑ linear time → O(n)
- ✓ McCreight's algorithm (1976)
  - ❑ linear time → O(n)
  - ❑ memory efficient
- ✓ Ukkonen's algorithm (1995)
  - ❑ linear time → O(n)
  - ❑ online

95

Runtime and Memory Analysis

- Runtime:
  - O(|*Genome*|) to construct the suffix tree ***directly.***
  - O(|*Genome*| + |*Patterns*|) to find pattern matches.
  - **Total: O(|*Genome*| + |*Patterns*|)**
- Memory:
  - O(|*Genome*|) to construct the suffix tree ***directly.***
  - O(|*Genome*|) to store the suffix tree.
  - **Total: O(|*Genome*| + |*Patterns*|)**

96

We are Not Finished Yet

- I am happy with the suffix tree, but I am not completely satisfied.
  - Runtime: O(|*Genome*| + |*Patterns*|)
  - Memory: O(|*Genome*|)
- However, big-O notation ignores constants!
  - The best known suffix tree implementations require ~ 20 times the length of |*Genome*|.
  - Can we reduce this constant factor?

97

From Theory to Practice

We are trained to see an algorithm with O(|*Text*|) runtime and memory and celebrate.

But a data structure with 2*|*Text*| memory is very different from one with 1000*|*Text*| memory.

It takes ~60 GB to store a suffix tree for a 3 GB human genome (4 bytes for each edge of tree). Can we do better?

98

SUFFIX ARRAYS

99

Suffix Array of “panamabananas$”

20 years go by...

Suffix array: Starting positions of sorted suffixes of *Text*.

Sorted Suffixes	Suffix Array
$	13
abananas$	5
amabananas$	3
anamabananas$	1
ananas$	7
anas$	9
as$	11
bananas$	6
mabananas$	4
namabananas$	2
nanas$	8
nas$	10
panamabananas$	0
s$	12

100

Suffix Array of "panamabananas$"

20 years go by...

Suffix array: Starting positions of sorted suffixes of *Text*.

Can be constructed in linear time, and takes only about 12 GB for human genome.

Sorted Suffixes	Suffix Array
$	13
abananas$	5
amabananas$	3
anamabananas$	1
ananas$	7
anas$	9
as$	11
bananas$	6
mabananas$	4
namabananas$	2
nanas$	8
nas$	10
panamabananas$	0
s$	12

101

Suffix Array of "panamabananas$"

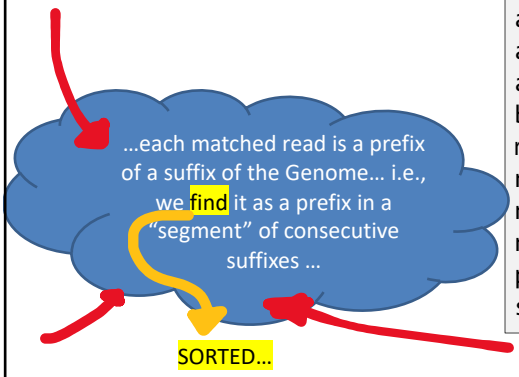
fattorizzazione di Lyndon per costruire il suffix array

Can be constructed in linear time, and takes only about 12 GB for human genome.

102

Suffix Array of "panamabananas$"

STOP: How can we use the suffix array for pattern matching?



Sorted Suffixes	Suffix Array
$	13
abananas$	5
amabananas$	3
anamabananas$	1
ananas$	7
anas$	9
as$	11
bananas$	6
mabananas$	4
namabananas$	2
nanas$	8
nas$	10
panamabananas$	0
s$	12

103

Suffix Array of "panamabananas$"

STOP: How can we use the suffix array for pattern matching?

Answer: we can pick a value, obtain "higher" or "lower", and move the next value to the middle of the unsearched suffixes.

Sorted Suffixes	Suffix Array
$	13
abananas$	5
amabananas$	3
anamabananas$	1
ananas$	7
anas$	9
as$	11
bananas$	6
mabananas$	4
namabananas$	2
nanas$	8
nas$	10
panamabananas$	0
s$	12

104

Suffix Array of "panamabananas$"

This approach, shown for *Pattern* = "ana", is the **binary search**.

Sorted Suffixes	Suffix Array
$	13
abananas$	5
amabananas$	3
anamabananas$	1
ananas$	7
anas$	9
as$	11
bananas$	6
mabananas$	4
namabananas$	2
nanas$	8
nas$	10
panamabananas$	0
s$	12

105

Suffix Array of "panamabananas$"

This approach, shown for *Pattern* = "ana", is the **binary search**.

First, we find the *first* occurrence of "ana".

Sorted Suffixes	Suffix Array
$	13
abananas$	5
amabananas$	3
anamabananas$	1
ananas$	7
anas$	9
as$	11
bananas$	6
mabananas$	4
namabananas$	2
nanas$	8
nas$	10
panamabananas$	0
s$	12

106

Suffix Array of "panamabananas$"

This approach, shown for *Pattern* = "ana", is the **binary search**.

First, we find the *first* occurrence of "ana".

Sorted Suffixes	Suffix Array
$	13
abananas$	5
amabananas$	3
anamabananas$	1
ananas$	7
anas$	9
as$	11
bananas$	6
mabananas$	4
namabananas$	2
nanas$	8
nas$	10
panamabananas$	0
s$	12

107

Suffix Array of "panamabananas$"

This approach, shown for *Pattern* = "ana", is the **binary search**.

First, we find the *first* occurrence of "ana".

Sorted Suffixes	Suffix Array
$	13
abananas$	5
amabananas$	3
anamabananas$	1
ananas$	7
anas$	9
as$	11
bananas$	6
mabananas$	4
namabananas$	2
nanas$	8
nas$	10
panamabananas$	0
s$	12

108

Suffix Array of "panamabananas$"

This approach, shown for *Pattern* = "ana", is the **binary search**.

First, we find the *first* occurrence of "ana".

Sorted Suffixes	Suffix Array
$	13
abananas$	5
amabananas$	3
anamabananas$	1
ananas$	7
anas$	9
as$	11
bananas$	6
mabananas$	4
namabananas$	2
nanas$	8
nas$	10
panamabananas$	0
s$	12

109

Suffix Array of "panamabananas$"

This approach, shown for *Pattern* = "ana", is the **binary search**.

First, we find the *first* occurrence of "ana".

Sorted Suffixes	Suffix Array
$	13
abananas$	5
amabananas$	3
anamabananas$	1
ananas$	7
anas$	9
as$	11
bananas$	6
mabananas$	4
namabananas$	2
nanas$	8
nas$	10
panamabananas$	0
s$	12

110

Suffix Array of “panamabananas$”

This approach, shown for *Pattern* = “ana”, is the **binary search**.

First, we find the *first* occurrence of “ana”.

Sorted Suffixes	Suffix Array
$	13
abananas$	5
amabananas$	3
anamabananas$	1
ananas$	7
anas$	9
as$	11
bananas$	6
mabananas$	4
namabananas$	2
nanas$	8
nas$	10
panamabananas$	0
s$	12

111

Suffix Array of “panamabananas$”

This approach, shown for *Pattern* = “ana”, is the **binary search**.

First, we find the *first* occurrence of “ana”.

Next, we find the *last* occurrence of “ana”.

Sorted Suffixes	Suffix Array
$	13
abananas$	5
amabananas$	3
anamabananas$	1
ananas$	7
anas$	9
as$	11
bananas$	6
mabananas$	4
namabananas$	2
nanas$	8
nas$	10
panamabananas$	0
s$	12

112

Suffix Array of “panamabananas$”

This approach, shown for *Pattern* = “ana”, is the **binary search**.

First, we find the *first* occurrence of “ana”.

Next, we find the *last* occurrence of “ana”.

Sorted Suffixes	Suffix Array
$	13
abananas$	5
amabananas$	3
anamabananas$	**1**
ananas$	7
anas$	9
as$	11
bananas$	6
mabananas$	**4**
namabananas$	2
nanas$	8
nas$	10
panamabananas$	0
s$	12

113

Suffix Array of “panamabananas$”

This approach, shown for *Pattern* = “ana”, is the **binary search**.

First, we find the *first* occurrence of “ana”.

Next, we find the *last* occurrence of “ana”.

Sorted Suffixes	Suffix Array
$	13
abananas$	5
amabananas$	3
anamabananas$	1
ananas$	7
anas$	**9**
as$	11
bananas$	6
mabananas$	**4**
namabananas$	2
nanas$	8
nas$	10
panamabananas$	0
s$	12

114

Suffix Array of "panamabananas$"

This approach, shown for *Pattern* = "ana", is the **binary search**.

First, we find the *first* occurrence of "ana".

Next, we find the *last* occurrence of "ana".

Sorted Suffixes	Suffix Array
$	13
abananas$	5
amabananas$	3
anamabananas$	1
ananas$	7
anas$	**9**
as$	**11**
bananas$	6
mabananas$	4
namabananas$	2
nanas$	8
nas$	10
panamabananas$	0
s$	12

115

Suffix Array of "panamabananas$"

This approach, shown for *Pattern* = "ana", is the **binary search**.

STOP: How do we know the starting locations of these three occurrences?

Sorted Suffixes	Suffix Array
$	13
abananas$	5
amabananas$	3
anamabananas$	**1**
ananas$	**7**
anas$	**9**
as$	11
bananas$	6
mabananas$	4
namabananas$	2
nanas$	8
nas$	10
panamabananas$	0
s$	12

116

Suffix Array of "panamabananas$"

This approach, shown for *Pattern* = "ana", is the **binary search**.

Answer: The suffix array tells us!

Sorted Suffixes	Suffix Array
$	13
abananas$	5
amabananas$	3
ana**mabananas$**	**1**
ana**nas$**	**7**
ana**s$**	**9**
as$	11
bananas$	6
mabananas$	4
namabananas$	2
nanas$	8
nas$	10
panamabananas$	0
s$	12

117

Suffix Array of "panamabananas$"

This approach, shown for *Pattern* = "ana", is the **binary search**.

STOP: What is the runtime of the binary search approach using the suffix array?

Sorted Suffixes	Suffix Array
$	13
abananas$	5
amabananas$	3
ana**mabananas$**	**1**
ana**nas$**	**7**
ana**s$**	**9**
as$	11
bananas$	6
mabananas$	4
namabananas$	2
nanas$	8
nas$	10
panamabananas$	0
s$	12

118

Suffix Array of "panamabananas$"

This approach, shown for *Pattern* = "ana", is the **binary search**.

Answer: Each pattern takes O(log(|*Text*|)).

Sorted Suffixes	Suffix Array
$	13
abananas$	5
amabananas$	3
anamabananas$	1
ananas$	7
anas$	9
as$	11
bananas$	6
mabananas$	4
namabananas$	2
nanas$	8
nas$	10
panamabananas$	0
s$	12

119

Binary Search Pseudocode

```
PatternMatchingSuffixArray(Text, Pattern)
        s ← SuffixArray(Text)
        minIndex ← 0
        maxIndex ← |Text| - 1
        while minIndex <= maxIndex
                midIndex ← floor((minIndex + maxIndex)/2)
                if Pattern > suffix of Text starting at s(midIndex)
                        minIndex ← midIndex + 1
                else
                        maxIndex ← midIndex - 1
        if Pattern matches suffix of Text starting at s(midIndex)
                first ← minIndex
        else
                return no occurrences
        minIndex ← first
        maxIndex ← |Text| - 1
        while minIndex <= maxIndex
                midIndex ← floor((minIndex + maxIndex)/2)
                if Pattern matches suffix of Text starting at s(midIndex)
                        minIndex ← midIndex + 1
                else
                        maxIndex ← midIndex – 1
        last ← maxIndex
        return (first, last)
```

120

Suffix Arrays are Another Example of Computational Biologists Influencing CS

Suffix arrays: a new method for on-line string searches
U Manber, G Myers - siam Journal on Computing, 1993 - SIAM
A new and conceptually simple data structure, called a suffix array, for on-line string searches is introduced in this paper. Constructing and querying suffix arrays is reduced to a sort and search paradigm that employs novel algorithms. The main advantage of suffix arrays over suffix trees is that, in practice, they use three to five times less space. From a complexity standpoint, suffix arrays permit on-line string searches of the type,"Is W a substring of A?" to be answered in time O(P+\logN), where P is the length of W and N is the …
☆ Cited by 2870 Related articles All 43 versions

Note: This is a paper produced by a computational biologist (and future head of search for Google) in a non-biological context seven years before the publication of the first draft human genome.

121

Suffix Arrays are Another Example of Computational Biologists Influencing CS

Suffix arrays: a new method for on-line string searches
U Manber, G Myers - siam Journal on Computing, 1993 - SIAM
A new and conceptually simple data structure, called a suffix array, for on-line string searches is introduced in this paper. Constructing and querying suffix arrays is reduced to a sort and search paradigm that employs novel algorithms. The main advantage of suffix arrays over suffix trees is that, in practice, they use three to five times less space. From a complexity standpoint, suffix arrays permit on-line string searches of the type "Is W a substring … d N is the …
☆

Kärkkäinen & Sanders (2003)
Suffix Array del testo T in tempo O(n)

Note: This is a paper produced by a computational biologist (and future head of search for Google) in a non-biological context seven years before the publication of the first draft human genome.

122

FATTORIZZAZIONI DI LYNDON E SA…

123

Outline

- Why do we map reads?
- Using the Trie
- From a Trie to a Suffix Tree
- **String Compression and the Burrows-Wheeler Transform**
- Inverting Burrows-Wheeler
- Using Burrows-Wheeler for Pattern Matching
- Finding the Matched Patterns
- Setting Up Checkpoints
- Inexact Matching
- Further Applications of Read Mapping

124

O(|G|) … Genome Compression

- Idea: decrease the amount of memory required to hold *Genome*.
- This indicates that we need methods of **compressing** a large genome, which is seemingly a separate problem.

125

Idea #1: Run-Length Encoding

- **Run-length encoding:** compresses a run of *n* identical symbols.

Genome

GGGGGGGGGGCCCCCCCCCCCCAAAAAAATTTTTTTTTTTTTTTTCCCCCG

10G11C7A15T5C1G

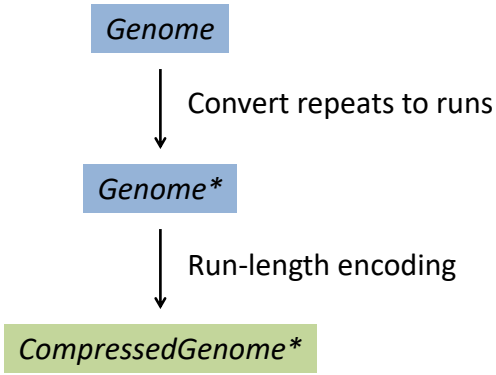
Run-length encoding

- Problem: Genomes don't have lots of runs…

126

Converting Repeats to Runs

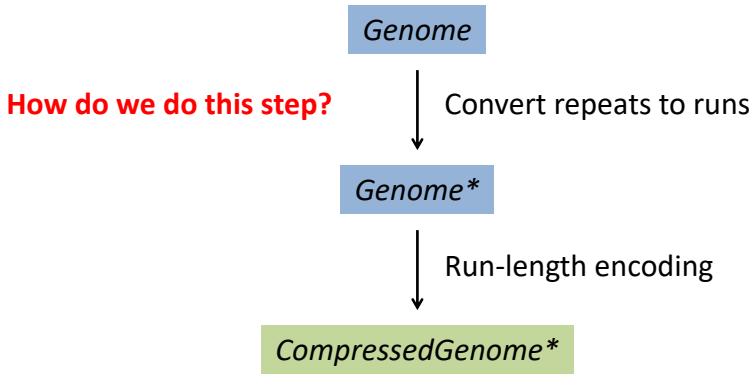
- …but they do have lots of repeats!



127

Converting Repeats to Runs

- …but they do have lots of repeats!



128

The Burrows-Wheeler Transform

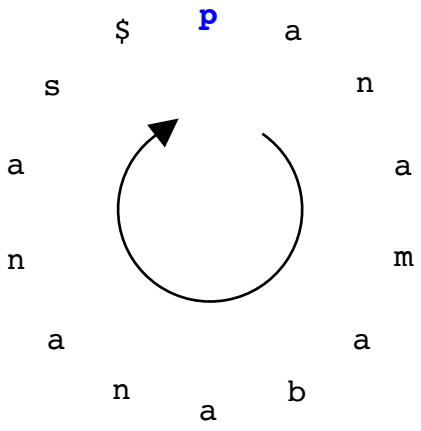
spiegata nella prima parte del corso, dove sono precisamente riportate:
- definizioni
- algoritmi
- correttezza degli algoritmi

ne vedremo solo l'uso per il mapping

129

The Burrows-Wheeler Transform

panamabananas$

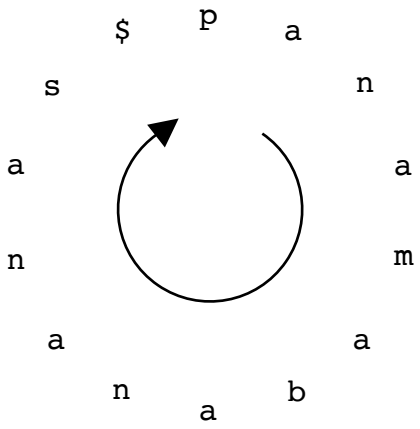


Form all cyclic rotations of "panamabananas$"

130

The Burrows-Wheeler Transform

panamabananas$
$panamabananas
s$panamabanana
as$panamabanan
nas$panamabana
anas$panamaban
nanas$panamaba
ananas$panamab
bananas$panama
abananas$panam
mabananas$pana
amabananas$pan
namabananas$pa
anamabananas$p



Form all cyclic rotations of "panamabananas$"

131

The Burrows-Wheeler Transform

panamabananas$
$panamabananas
s$panamabanana
as$panamabanan
nas$panamabana
anas$panamaban
nanas$panamaba
ananas$panamab
bananas$panama
abananas$panam
mabananas$pana
amabananas$pan
namabananas$pa
anamabananas$p

$panamabananas
abananas$panam
amabananas$pan
anamabananas$p
ananas$panamab
anas$panamaban
as$panamabanan
bananas$panama
mabananas$pana
namabananas$pa
nanas$panamaba
nas$panamabana
panamabananas$
s$panamabanana

Form all cyclic rotations of "panamabananas$"

Sort the strings lexicographically ($ comes first)
the color shows that the string are sorted

132

The Burrows-Wheeler Transform

panamabananas$	$panamabananas**s**
$panamabananas	abananas$panam**m**
s$panamabanana	amabananas$pan**n**
as$panamabanan	anamabananas$**p**
nas$panamabana	ananas$panama**b**
anas$panamaban	anas$panamaba**n**
nanas$panamaba	as$panamabana**n**
ananas$panamab	bananas$panam**a**
bananas$panama	mabananas$pan**a**
abananas$panam	namabananas$p**a**
mabananas$pana	nanas$panamab**a**
amabananas$pan	nas$panamaban**a**
namabananas$pa	panamabananas**$**
anamabananas$p	s$panamabanan**a**

Form all cyclic rotations of "panamabananas$"

Burrows-Wheeler Transform: Last column = smnpbnnaaaaa$a

133

the LAST column… Why Should We Care?

Watson-Crick paper on double structure of DNA

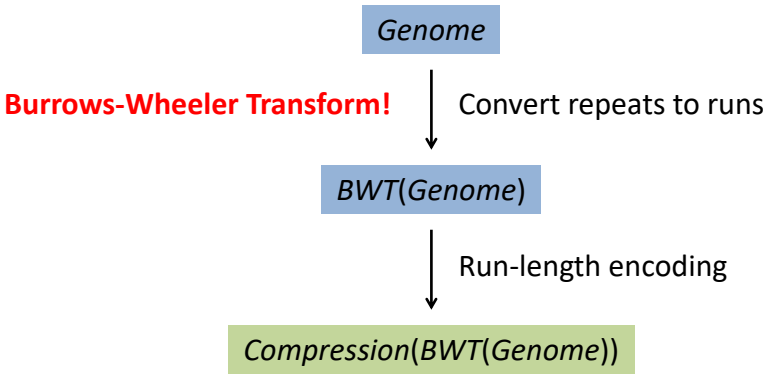
```
nd Corey (1).  They kindly made their manuscript availa ..... a
nd criticism, especially on interatomic distances.  We  ..... a
nd cytosine.  The sequence of bases on a single chain d ..... a
nd experimentally (3,4) that the ratio of the amounts o ..... u
nd for this reason we shall not comment on it.  We wish ..... a
nd guanine (purine) with cytosine (pyrimidine).  In oth ..... a
nd ideas of Dr.  M. H. F. Wilkins, Dr.  R. E. Franklin  ..... a
nd its water content is rather high.  At lower water co ..... a
nd pyrimidine bases.  The planes of the bases are perpe ..... a
nd stereochemical arguments.  It has not escaped our no ..... a
nd that only specific pairs of bases can bond together  ..... u
nd the atoms near it is close to Furberg's 'standard co ..... a
nd the bases on the inside, linked together by hydrogen ..... a
nd the bases on the outside.  In our opinion, this stru ..... a
nd the other a pyrimidine for bonding to occur.  The hy ..... a
nd the phosphates on the outside.  The configuration of ..... a
nd the ration of guanine to cytosine, are always very c ..... a
nd the same axis (see diagram).  We have made the usual ..... u
nd their co-workers at King's College, London.  One of  ..... a
```

qui no…

the text has a lot of repetitive words… (the word «and»)

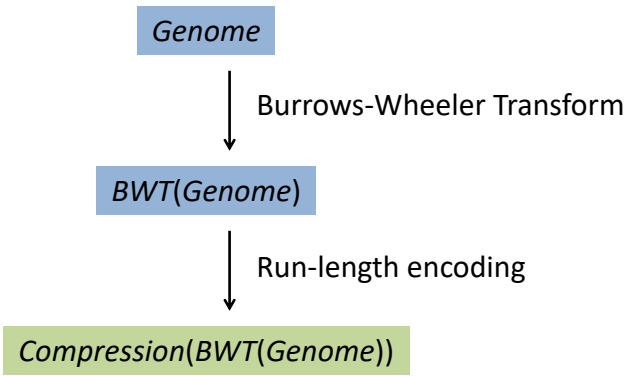
134

BWT: Converting Repeats to Runs



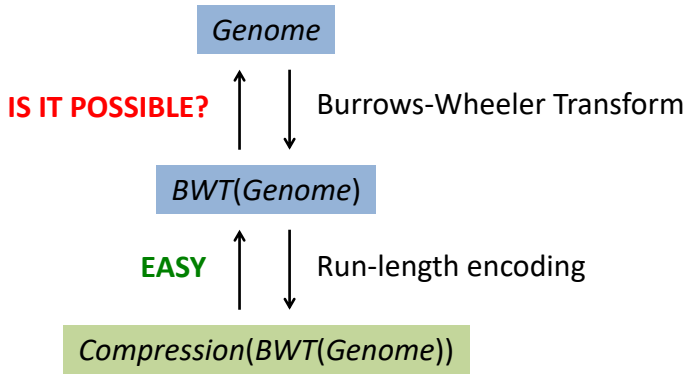
135

How Can We Decompress?



136

How Can We Decompress?



137

Outline

- Why do we map reads?
- Using the Trie
- From a Trie to a Suffix Tree
- String Compression and the Burrows-Wheeler Transform
- **Inverting Burrows-Wheeler**
- Using Burrows-Wheeler for Pattern Matching
- Finding the Matched Patterns
- Setting Up Checkpoints
- Inexact Matching
- Further Applications of Read Mapping

138

Reconstructing banana

```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```

We have the last column, we obtain the string

139

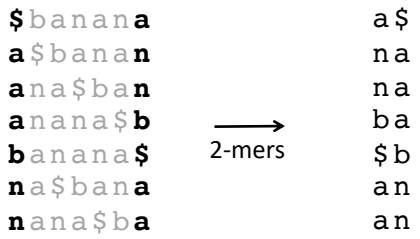
Reconstructing banana

```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```

- Sorting all elements of “annb$aa” gives first column of matrix [since the strings are lexicographically sorted]

140

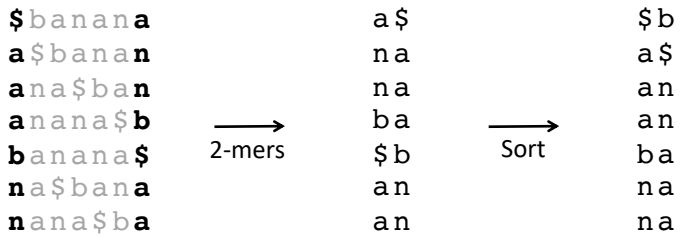
Reconstructing banana



- We now know 2-mer composition of the circular string banana$

141

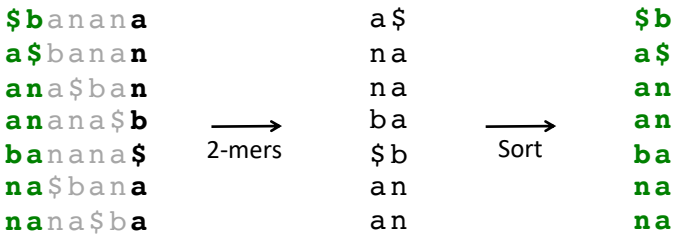
Reconstructing banana



- We now know 2-mer composition of the circular string banana$
- Sorting gives us the first 2 columns of the matrix.

142

Reconstructing banana



- We now know 2-mer composition of the circular string banana$
- Sorting gives us the first 2 columns of the matrix.

143

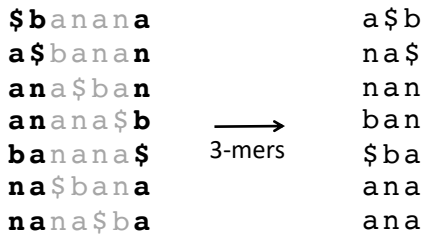
Reconstructing banana

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

We use the first two columns + the last column, and we obtain the 3-mer composition ot the circular string «banana»

144

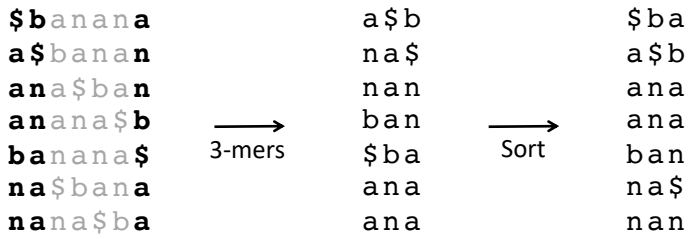
Reconstructing banana



- We now know 3-mer composition of the circular string banana$

145

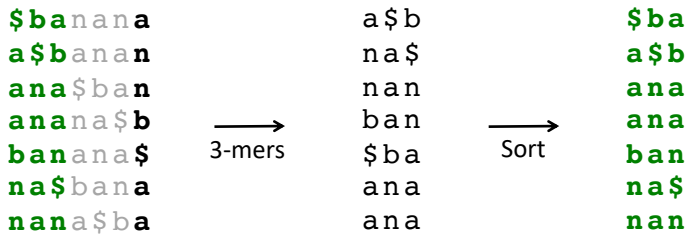
Reconstructing banana



- We now know 3-mer composition of the circular string banana$
- Sorting gives us the first 3 columns of the matrix.

146

Reconstructing banana



- We now know 3-mer composition of the circular string banana$
- Sorting gives us the first 3 columns of the matrix.

147

Reconstructing banana

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

148

Reconstructing banana

$banan**a**		a$ba
a$banan**n**		na$b
ana$ba**n**		nana
anana$**b**	⟶ 4-mers	bana
banana**$**		$ban
na$ban**a**		ana$
nana$b**a**		anan

- We now know 4-mer composition of the circular string banana$

149

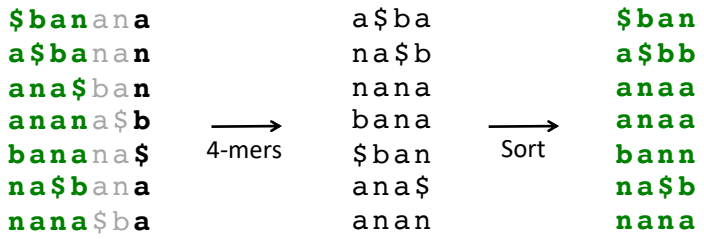
Reconstructing banana

$banan**a**		a$ba		$ban
a$banan**n**		na$b		a$bb
ana$ba**n**		nana		anaa
anana$**b**	⟶ 4-mers	bana	⟶ Sort	anaa
banana**$**		$ban		bann
na$ban**a**		ana$		na$b
nana$b**a**		anan		nana

- We now know 4-mer composition of the circular string banana$
- Sorting gives us the first 4 columns of the matrix.

150

Reconstructing banana



- We now know 4-mer composition of the circular string banana$
- Sorting gives us the first 4 columns of the matrix.

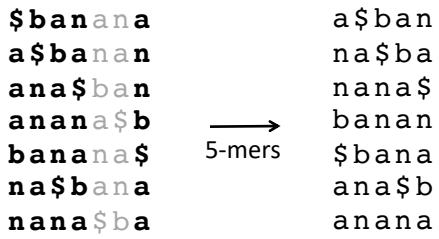
151

Reconstructing banana

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

152

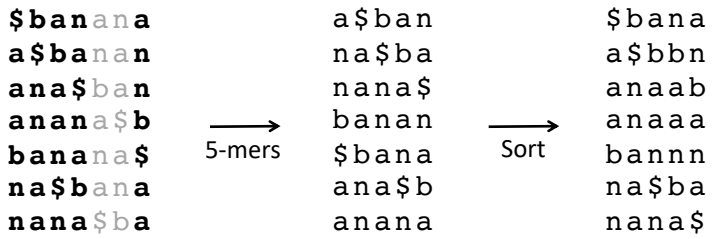
Reconstructing banana



- We now know 5-mer composition of the circular string banana$

153

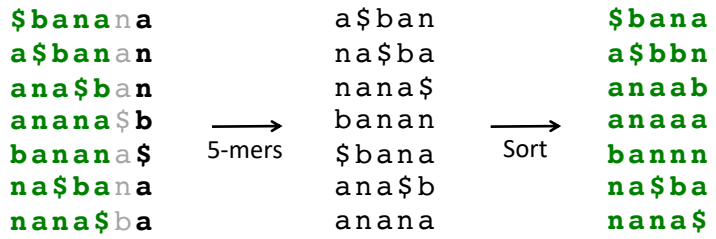
Reconstructing banana



- We now know 5-mer composition of the circular string banana$
- Sorting gives us the first 5 columns of the matrix.

154

Reconstructing banana



- We now know 5-mer composition of the circular string banana$
- Sorting gives us the first 5 columns of the matrix.

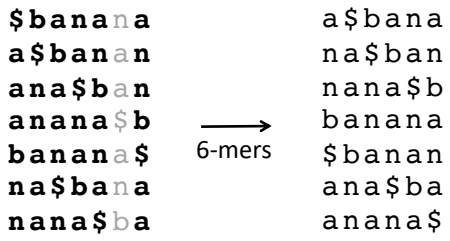
155

Reconstructing banana

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

156

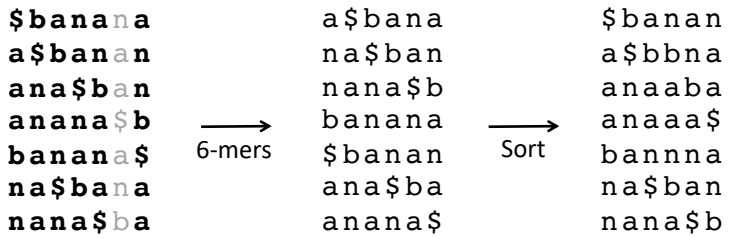
Reconstructing banana



- We now know 6-mer composition of the circular string banana$

157

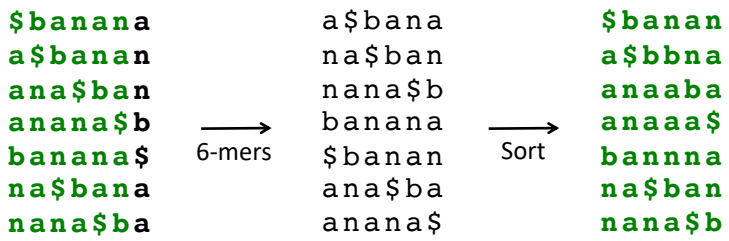
Reconstructing banana



- We now know 6-mer composition of the circular string banana$
- Sorting gives us the first 6 columns of the matrix.

158

Reconstructing banana



- We now know 6-mer composition of the circular string banana$
- Sorting gives us the first 6 columns of the matrix.

159

Reconstructing banana

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

- We now know the entire matrix!

160

Reconstructing banana

```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```

- We now know the entire matrix!
- Taking all elements in the first row (after $) produces banana.

161

More Memory Issues

- Reconstructing *Genome* from *BWT*(*Genome*) required us to store |*Genome*| copies of |*Genome*|.

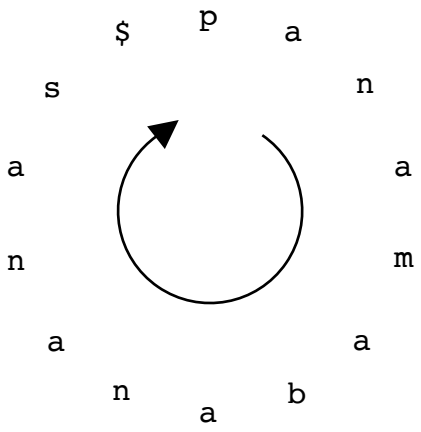
```
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba
```

- **Can we invert BWT with less space?**

162

A "Strange" Observation

```
$panamabananas
abananas$panam
amabananas$pan
anamabananas$p
ananas$panamab
anas$panamaban
as$panamabanan
bananas$panama
mabananas$pana
namabananas$pa
nanas$panamaba
nas$panamabana
panamabananas$
s$panamabanana
```

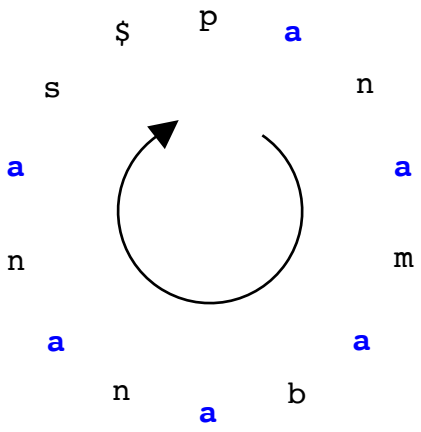


We have the last column, and the first column for free.

163

A Strange Observation

```
$panamabananas
abananas$panam
amabananas$pan
anamabananas$p
ananas$panamab
anas$panamaban
as$panamabanan
bananas$panama
mabananas$pana
namabananas$pa
nanas$panamaba
nas$panamabana
panamabananas$
s$panamabanana
```

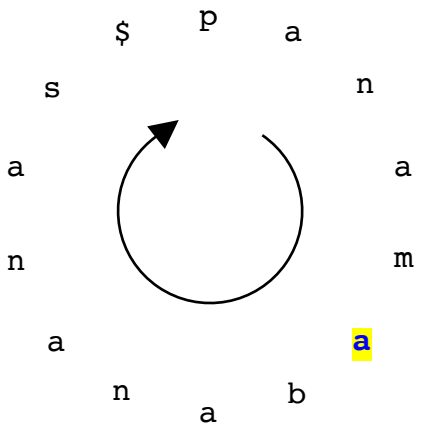


See the six occurrence of a in the first column, in the last column and in the circle representation

164

A Strange Observation

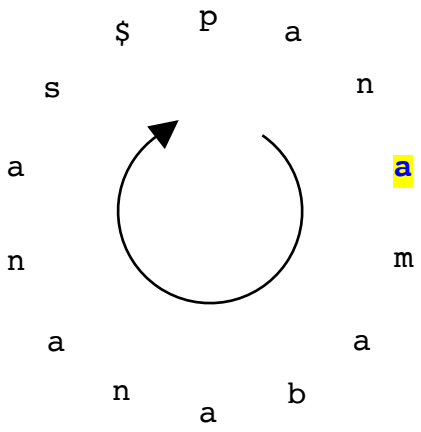
```
$panamabananas
abananas$panam
amabananas$pan
anamabananas$p
ananas$panamab
anas$panamaban
as$panamabanan
bananas$panama
mabananas$pana
namabananas$pa
nanas$panamaba
nas$panamabana
panamabananas$
s$panamabanana
```



165

A Strange Observation

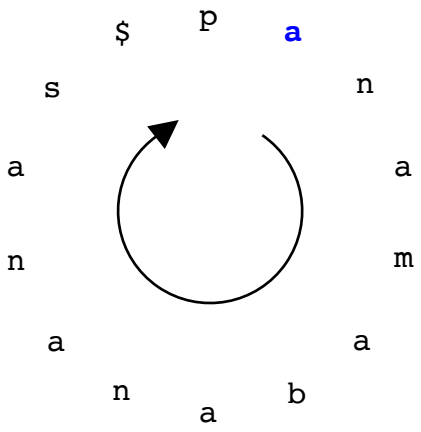
```
$panamabananas
abananas$panam
amabananas$pan
anamabananas$p
ananas$panamab
anas$panamaban
as$panamabanan
bananas$panama
mabananas$pana
namabananas$pa
nanas$panamaba
nas$panamabana
panamabananas$
s$panamabanana
```



166

A Strange Observation

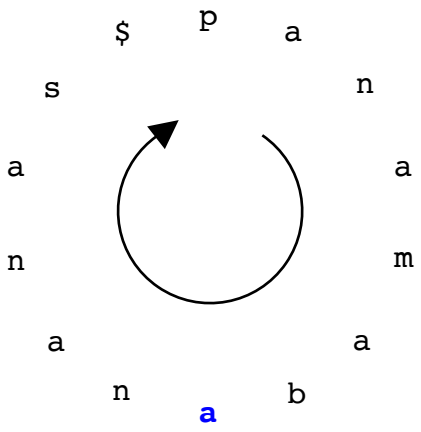
$panamabananas
abananas$panam
amabananas$pan
anamabananas$p
ananas$panamab
anas$panamaban
as$panamabanan
bananas$panama
mabananas$pana
namabananas$p**a**
nanas$panamaba
nas$panamabana
panamabananas$
s$panamabanana



167

A Strange Observation

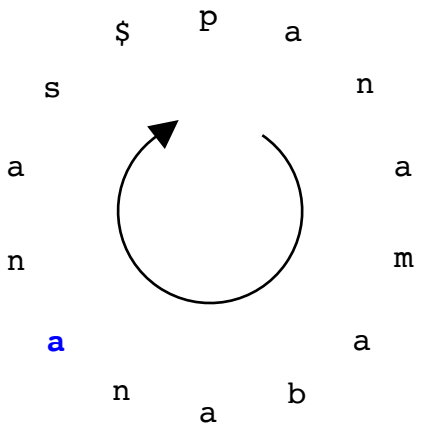
$panamabananas
abananas$panam
amabananas$pan
anamabananas$p
ananas$panamab
anas$panamaban
as$panamabanan
bananas$panama
mabananas$pana
namabananas$pa
nanas$panamab**a**
nas$panamabana
panamabananas$
s$panamabanana



168

A Strange Observation

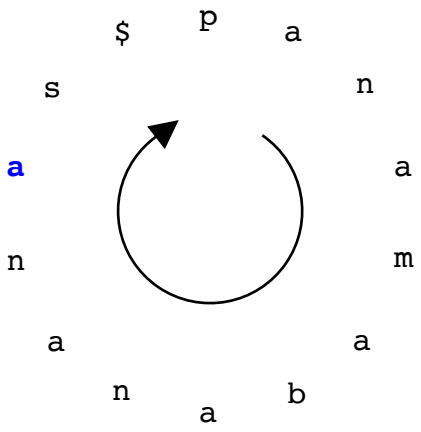
```
$panamabananas
abananas$panam
amabananas$pan
anamabananas$p
ananas$panamab
anas$panamaban
as$panamabanan
bananas$panama
mabananas$pana
namabananas$pa
nanas$panamaba
nas$panamabana
panamabananas$
s$panamabanana
```



169

A Strange Observation

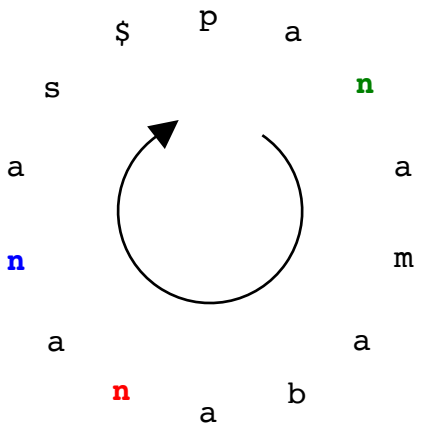
```
$panamabananas
abananas$panam
amabananas$pan
anamabananas$p
ananas$panamab
anas$panamaban
as$panamabanan
bananas$panama
mabananas$pana
namabananas$pa
nanas$panamaba
nas$panamabana
panamabananas$
s$panamabanana
```



170

A Strange Observation

```
$panamabananas
abananas$panam
amabananas$pan
anamabananas$p
ananas$panamab
anas$panamaban
as$panamabanan
bananas$panama
mabananas$pana
namabananas$pa
nanas$panamaba
nas$panamabana
panamabananas$
s$panamabanana
```



171

Is It True in General?

We want to find a faster way of inverting BWT

172

Is It True in General?

```
   $panamabananas
1  abananas$panam
2  amabananas$pan
3  anamabananas$p
4  ananas$panamab
5  anas$panamaban
6  as$panamabanan
   bananas$panama
   mabananas$pana
   namabananas$pa
   nanas$panamaba
   nas$panamabana
   panamabananas$
   s$panamabanana
```

These strings are sorted

173

Is It True in General?

```
   $panamabananas
1  abananas$panam
2  amabananas$pan
3  anamabananas$p
4  ananas$panamab
5  anas$panamaban
6  as$panamabanan
   bananas$panama
   mabananas$pana
   namabananas$pa
   nanas$panamaba
   nas$panamabana
   panamabananas$
   s$panamabanana
```

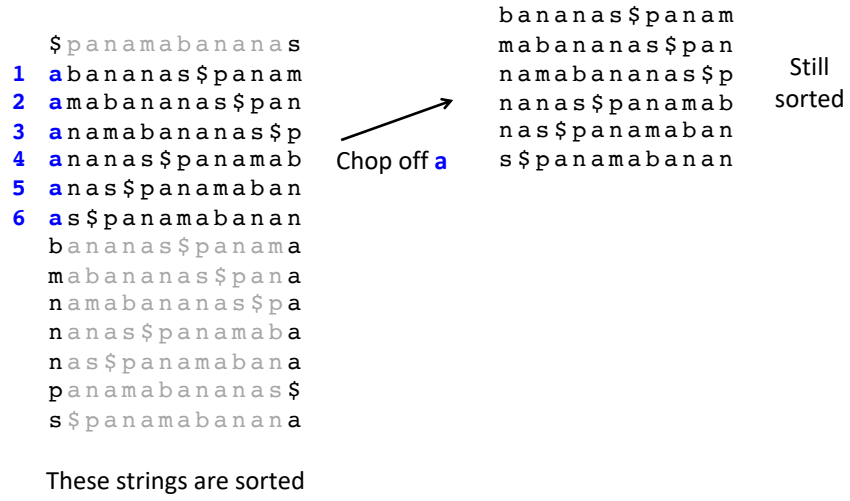
Chop off **a**

```
bananas$panam
mabananas$pan
namabananas$p
nanas$panamab
nas$panamaban
s$panamabanan
```

These strings are sorted

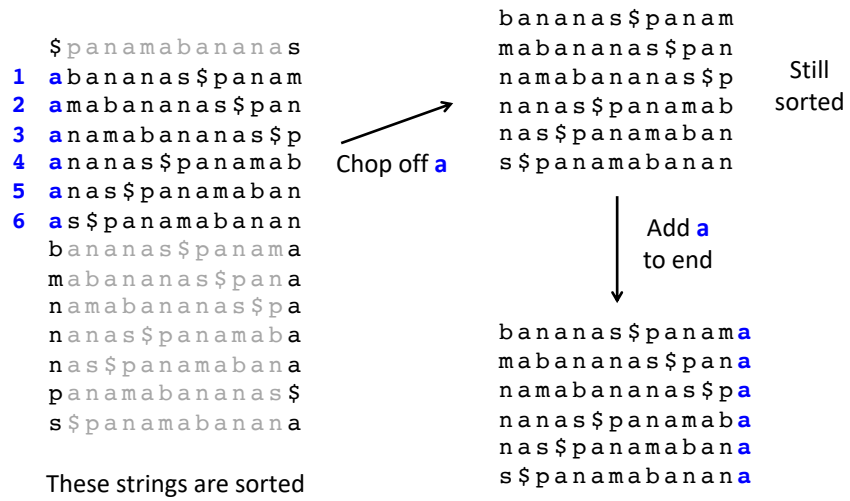
174

Is It True in General?



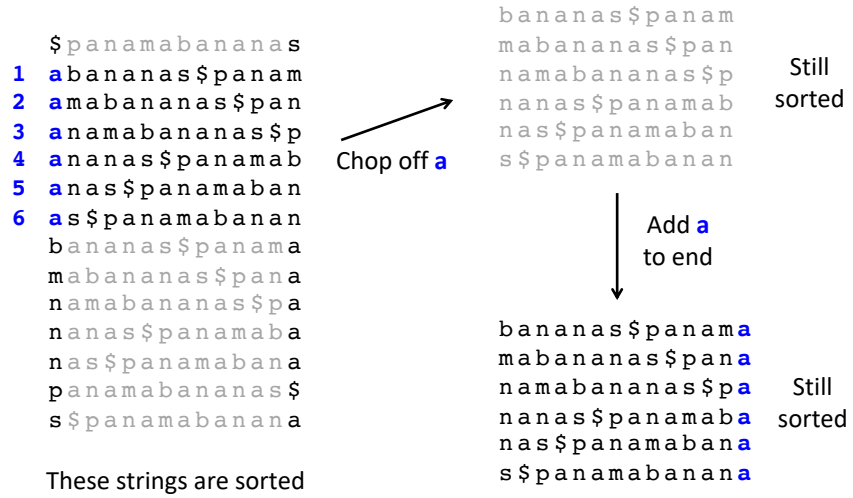
175

Is It True in General?



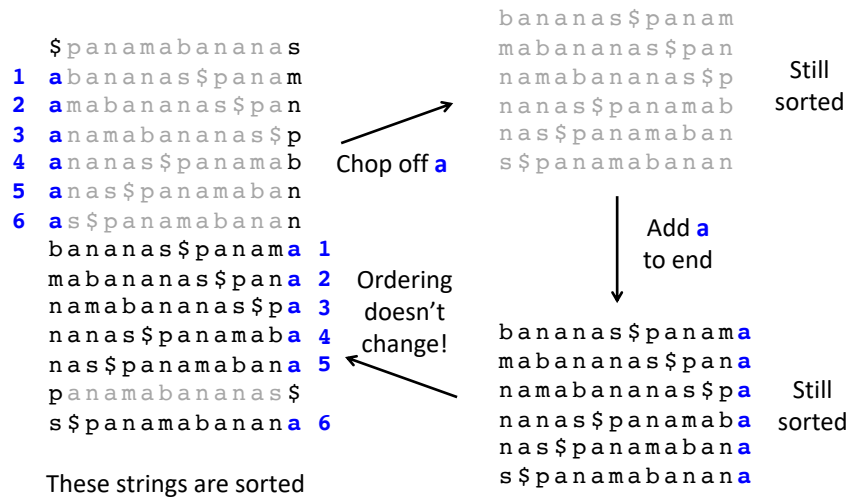
176

Is It True in General?



177

Is It True in General?



178

Is It True in General?

- **First-Last Property**: The *k*-th occurrence of *symbol* in *FirstColumn* and the *k*-th occurrence of *symbol* in *LastColumn* correspond to the same position of *symbol* in *Genome*.

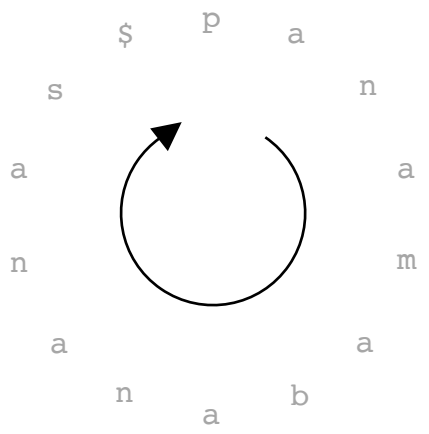
spiegata nella prima parte del corso, dove sono precisamente riportate:
- definizioni
- algoritmi
- correttezza degli algoritmi

$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$

179

More Efficient BWT Decompression

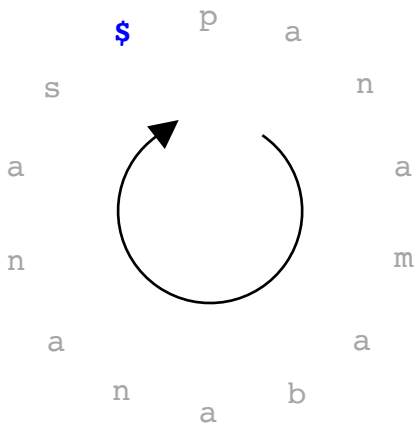
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



180

More Efficient BWT Decompression

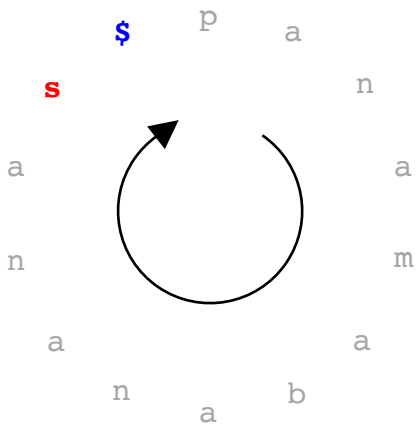
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



181

More Efficient BWT Decompression

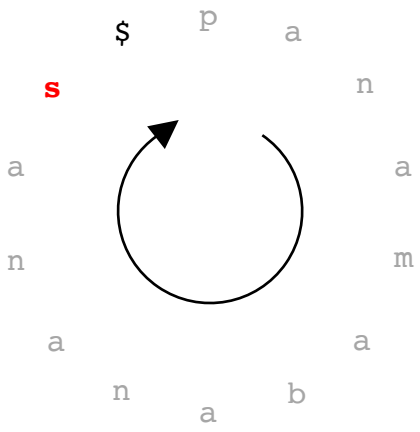
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



182

More Efficient BWT Decompression

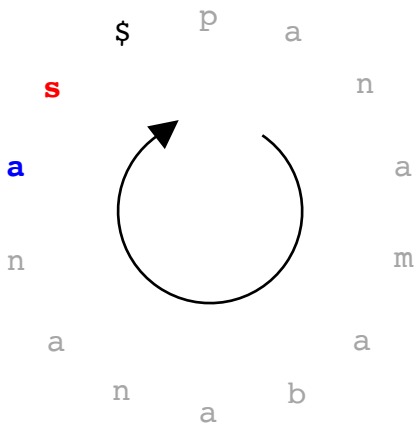
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



183

More Efficient BWT Decompression

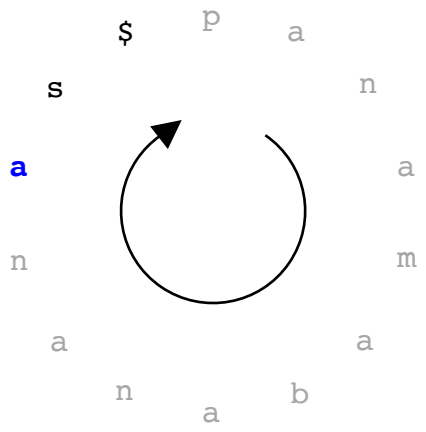
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



184

More Efficient BWT Decompression

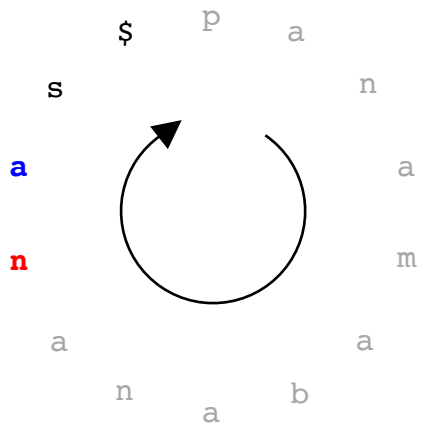
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanan**a$_6$**



185

More Efficient BWT Decompression

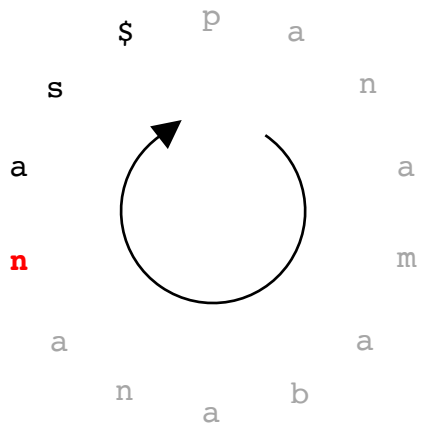
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan**n$_3$**
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



186

More Efficient BWT Decompression

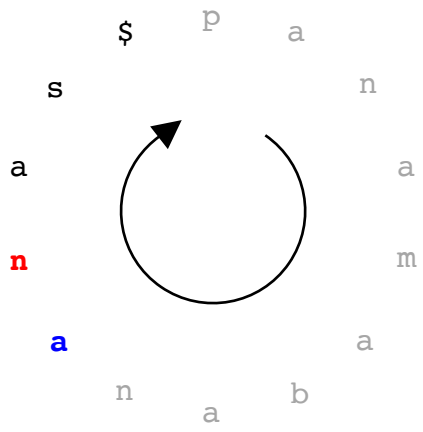
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



187

More Efficient BWT Decompression

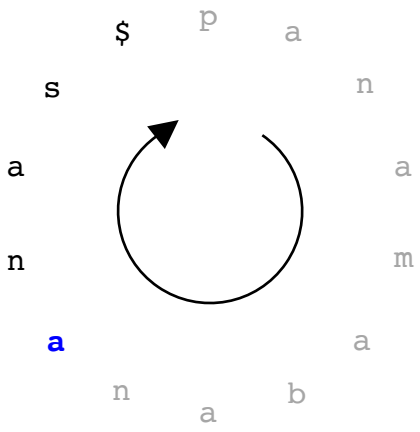
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



188

More Efficient BWT Decompression

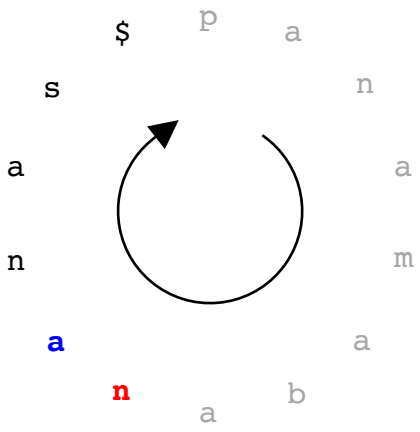
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



189

More Efficient BWT Decompression

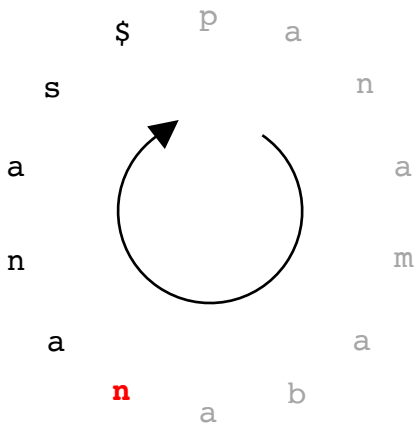
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



190

More Efficient BWT Decompression

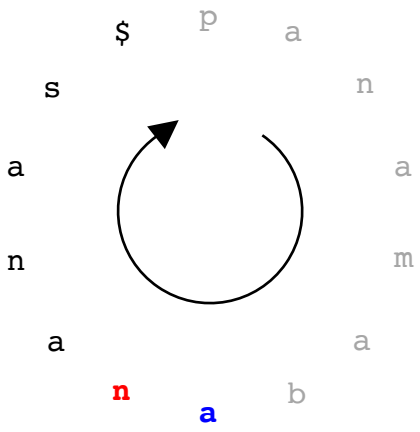
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



191

More Efficient BWT Decompression

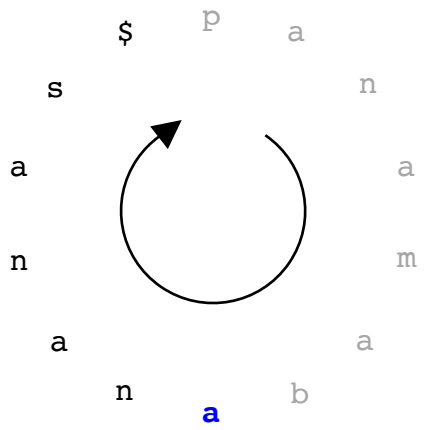
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



192

More Efficient BWT Decompression

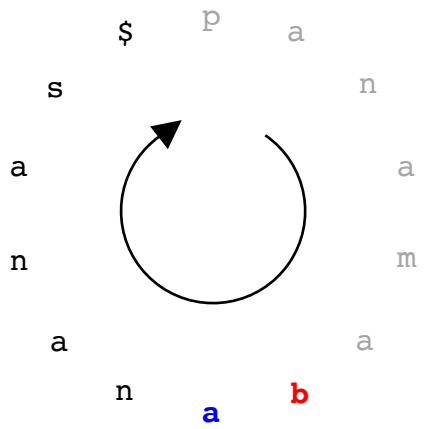
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamab**a$_4$**
n$_3$as$panamaban$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



193

More Efficient BWT Decompression

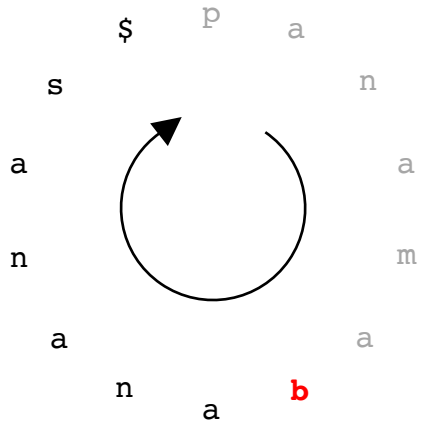
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panama**b$_1$**
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamab**a$_4$**
n$_3$as$panamaban$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



194

More Efficient BWT Decompression

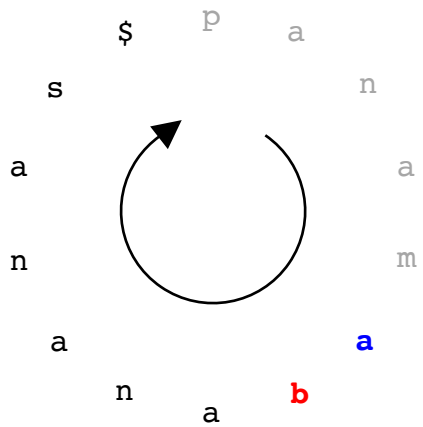
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



195

More Efficient BWT Decompression

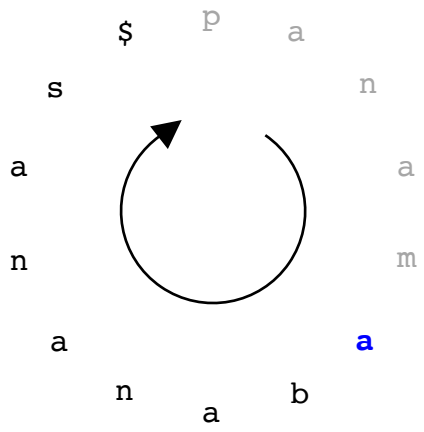
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



196

More Efficient BWT Decompression

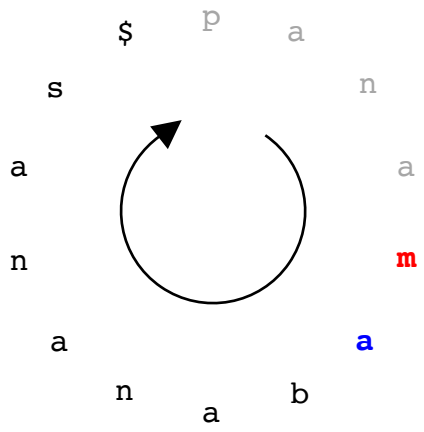
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panam**a$_1$**
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



197

More Efficient BWT Decompression

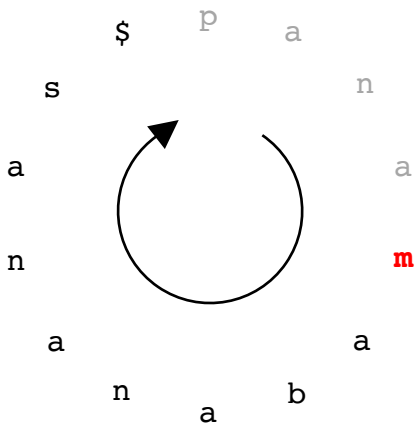
$_1$panamabananas$_1$
a$_1$bananas$panam**m$_1$**
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



198

More Efficient BWT Decompression

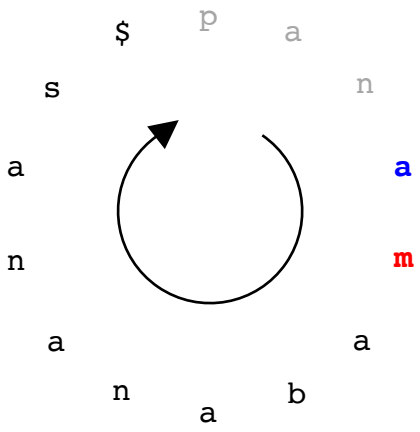
$_1$panamabananas$_1$
a$_1$bananas$panam**m**$_1$
a$_2$mabananas$pan**n**$_1$
a$_3$namabananas$p**p**$_1$
a$_4$nanas$panama**b**$_1$
a$_5$nas$panamaba**n**$_2$
a$_6$s$panamabana**n**$_3$
b$_1$ananas$panama**a**$_1$
m$_1$abananas$pana**a**$_2$
n$_1$amabananas$pa**a**$_3$
n$_2$anas$panamab**a**$_4$
n$_3$as$panamaban**a**$_5$
p$_1$anamabananas$**$**$_1$
s$_1$$panamabanan**a**$_6$



199

More Efficient BWT Decompression

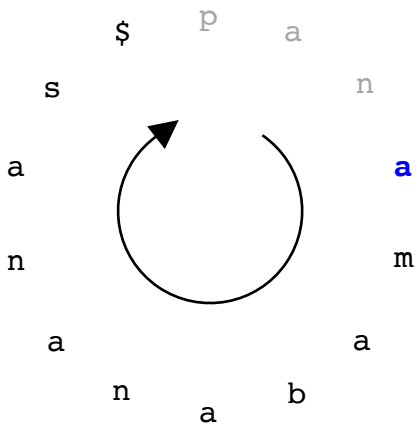
$_1$panamabananas$_1$
a$_1$bananas$panam**m**$_1$
a$_2$mabananas$pan**n**$_1$
a$_3$namabananas$p**p**$_1$
a$_4$nanas$panama**b**$_1$
a$_5$nas$panamaba**n**$_2$
a$_6$s$panamabana**n**$_3$
b$_1$ananas$panama**a**$_1$
m$_1$abananas$pan**a**$_2$
n$_1$amabananas$pa**a**$_3$
n$_2$anas$panamab**a**$_4$
n$_3$as$panamaban**a**$_5$
p$_1$anamabananas$**$**$_1$
s$_1$$panamabanan**a**$_6$



200

More Efficient BWT Decompression

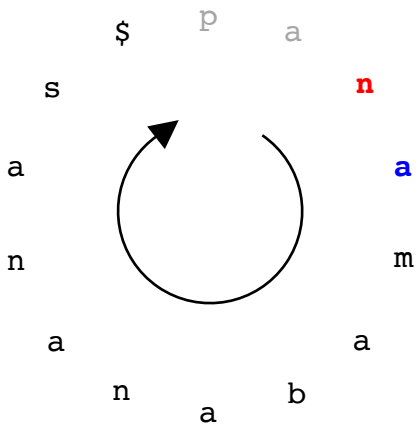
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



201

More Efficient BWT Decompression

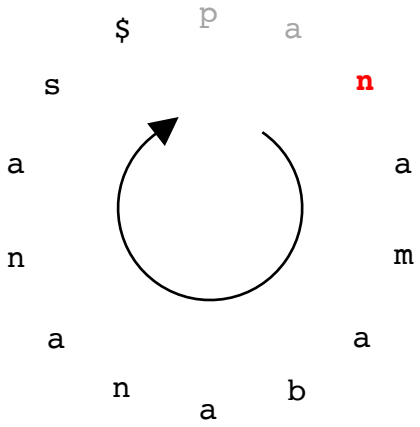
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



202

More Efficient BWT Decompression

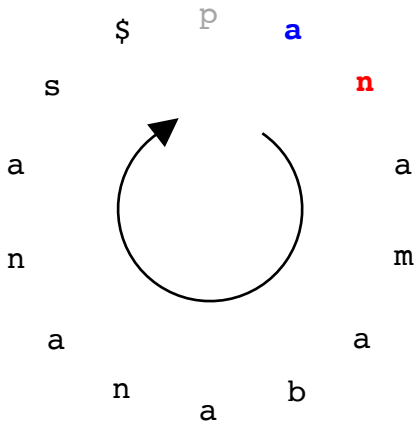
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananas$pan**n**$_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



203

More Efficient BWT Decompression

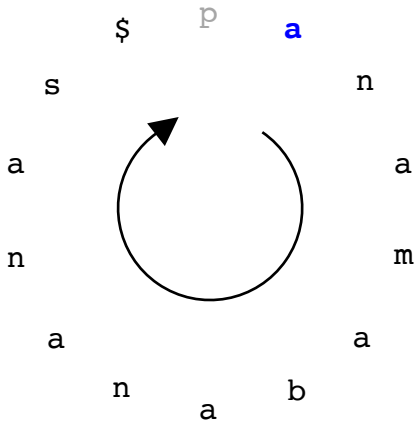
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananas$p**a**$_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



204

More Efficient BWT Decompression

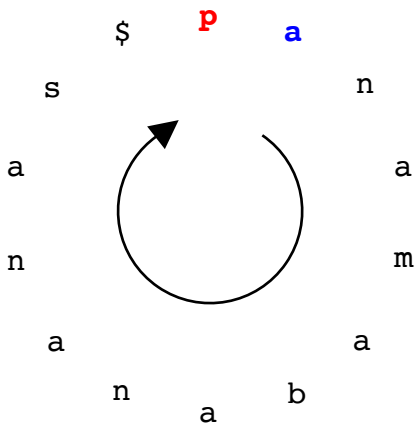
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananas$p**a$_3$**
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



205

More Efficient BWT Decompression

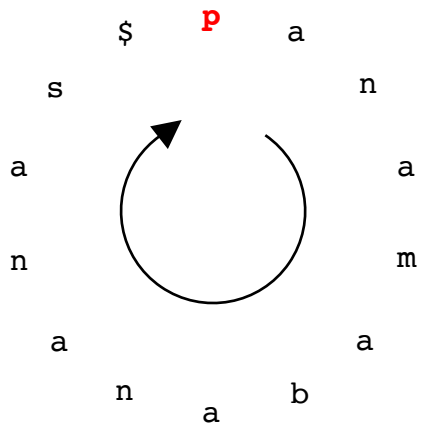
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$**p$_1$**
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



206

More Efficient BWT Decompression

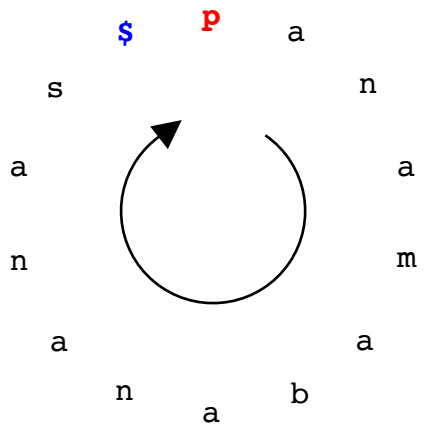
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



207

More Efficient BWT Decompression

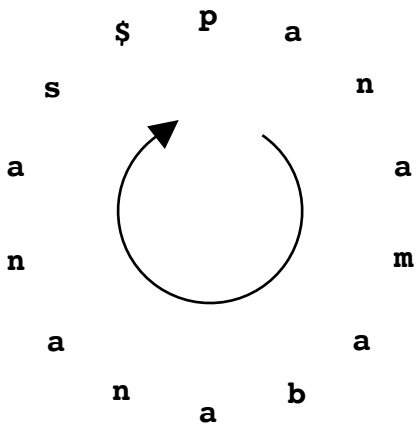
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



208

More Efficient BWT Decompression

$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$



- Memory: 2|*Genome*| = O(|*Genome*|).

209

What happened to pattern matching?

210

Outline

- Why do we map reads?
- Using the Trie
- From a Trie to a Suffix Tree
- String Compression and the Burrows-Wheeler Transform
- Inverting Burrows-Wheeler
- **Using Burrows-Wheeler for Pattern Matching**
- Finding the Matched Patterns
- Setting Up Checkpoints
- Inexact Matching
- Further Applications of Read Mapping

211

Recalling Our Goal

- Suffix Tree/array Pattern Matching:
  - Runtime: O(|*Genome*| + |*Patterns*|)
  - Memory: O(|*Genome*|)
  - Problem: suffix tree takes 20 x |*Genome*| space, but suffix array takes ~4 x |*Genome*| space.
- Can we use BWT(*Genome*) as our data structure instead? BWT is a column…

212

Finding Pattern Matches Using BWT

- Searching for **ana** in p**ana**mab**anana**s

$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$

213

Finding Pattern Matches Using BWT

- Searching for ana in panamabananas

$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$

214

Finding Pattern Matches Using BWT

- Searching for **ana** in panamabananas

$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$

215

Finding Pattern Matches Using BWT

- Searching for **ana** in panamabananas

$_1$panamabananas$_1$
a$_1$bananas$panam**$_1$
a$_2$mabananas$pan**$_1$
a$_3$namabananas$p**$_1$
a$_4$nanas$panamab**$_1$
a$_5$nas$panamaban**$_2$
a$_6$s$panamabanan**$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$

216

Finding Pattern Matches Using BWT

- Searching for **ana** in panamabananas

$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananas$pa**n$_1$**
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaba**n$_2$**
a$_6$s$panamabana**n$_3$**
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$

217

Finding Pattern Matches Using BWT

- Searching for **ana** in panamabananas

$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananas$pa**n**$_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaba**n**$_2$
a$_6$s$panamabana**n**$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$

218

Finding Pattern Matches Using BWT

- Searching for **ana** in panamabananas

$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$

219

Finding Pattern Matches Using BWT

- Searching for **ana** in panamabananas

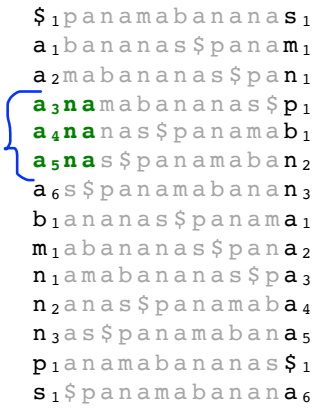
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananas$pa**$_3$**
n$_2$anas$panamaba**$_4$**
n$_3$as$panamabana**$_5$**
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$

220

Finding Pattern Matches Using BWT

- Searching for **ana** in panamabananas

consecutive occurrences!



221

Outline

- Why do we map reads?
- Using the Trie
- From a Trie to a Suffix Tree
- String Compression and the Burrows-Wheeler Transform
- Inverting Burrows-Wheeler
- Using Burrows-Wheeler for Pattern Matching
- **Finding the Matched Patterns**
- Setting Up Checkpoints
- Inexact Matching
- Further Applications of Read Mapping

222

Where Are the Matches?

- **Multiple Pattern Matching Problem:**
  - **Input:** A collection of strings *Patterns* and a string *Genome.*
  - **Output:** All positions in *Genome* where one of *Patterns* appears as a substring.

223

Where Are the Matches?

- **Multiple Pattern Matching Problem:**
  - **Input:** A collection of strings *Patterns* and a string *Genome.*
  - **Output:** All **positions** in *Genome* where one of *Patterns* appears as a substring.
- Where are the **positions**? BWT has not revealed them.

224

Where Are the Matches?

- Example: We know that **ana** occurs 3 times, but where?

$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$

225

Using the Suffix Array to Find Matches

- **Suffix array:** holds starting position of each suffix beginning a row.

From BWT to SA
(mentre costruisco la BWT, creo il SA)

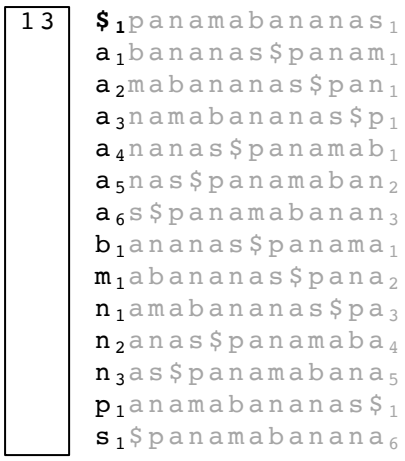
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamabana$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$

226

Using the Suffix Array to Find Matches

- **Suffix array:** holds starting position of each suffix beginning a row.

panamabananas**$**

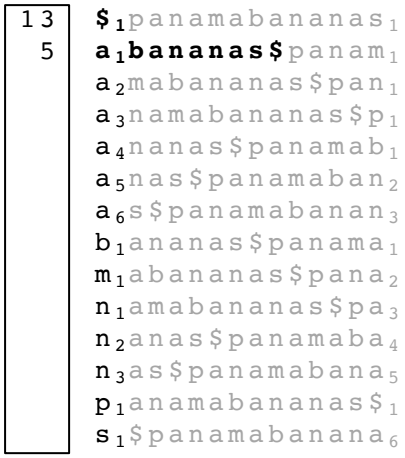


227

Using the Suffix Array to Find Matches

- **Suffix array:** holds starting position of each suffix beginning a row.

panam**abananas$**



228

Using the Suffix Array to Find Matches

- **Suffix array:** holds starting position of each suffix beginning a row.

pan**amabananas$**

13	**$$_1$**panamabananas$_1$
5	**a$_1$bananas$**panam$_1$
3	**a$_2$mabananas$**pan$_1$
	a$_3$namabananas$p$_1$
	a$_4$nanas$panamab$_1$
	a$_5$nas$panamaban$_2$
	a$_6$s$panamabanan$_3$
	b$_1$ananas$panama$_1$
	m$_1$abananas$pana$_2$
	n$_1$amabananaspa_3$
	n$_2$anas$panamaba$_4$
	n$_3$as$panamabana$_5$
	p$_1$anamabananas$$_1$
	s$_1$$panamabanana$_6$

229

Using the Suffix Array to Find Matches

- **Suffix array:** holds starting position of each suffix beginning a row.

p**anamabananas$**

13	**$$_1$**panamabananas$_1$
5	**a$_1$bananas$**panam$_1$
3	**a$_2$mabananas$**pan$_1$
1	**a$_3$namabananas$**p$_1$
	a$_4$nanas$panamab$_1$
	a$_5$nas$panamaban$_2$
	a$_6$s$panamabanan$_3$
	b$_1$ananas$panama$_1$
	m$_1$abananas$pana$_2$
	n$_1$amabananaspa_3$
	n$_2$anas$panamaba$_4$
	n$_3$as$panamabana$_5$
	p$_1$anamabananas$$_1$
	s$_1$$panamabanana$_6$

230

Using the Suffix Array to Find Matches

- **Suffix array:** holds starting position of each suffix beginning a row.

panamab**ananas$**

13	**$₁**panamabananas₁
5	**a₁bananas$**panam₁
3	**a₂mabananas$**pan₁
1	**a₃namabananas$**p₁
7	**a₄nanas$**panamab₁
	a₅nas$panamaban₂
	a₆s$panamabanan₃
	b₁ananas$panama₁
	m₁abananas$pana₂
	n₁amabananas$pa₃
	n₂anas$panamaba₄
	n₃as$panamabana₅
	p₁anamabananas$₁
	s₁$panamabanana₆

231

Using the Suffix Array to Find Matches

- **Suffix array:** holds starting position of each suffix beginning a row.

panamaban**anas$**

13	**$₁**panamabananas₁
5	**a₁bananas$**panam₁
3	**a₂mabananas$**pan₁
1	**a₃namabananas$**p₁
7	**a₄nanas$**panamab₁
9	**a₅nas$**panamaban₂
	a₆s$panamabanan₃
	b₁ananas$panama₁
	m₁abananas$pana₂
	n₁amabananas$pa₃
	n₂anas$panamaba₄
	n₃as$panamabana₅
	p₁anamabananas$₁
	s₁$panamabanana₆

232

Using the Suffix Array to Find Matches

- **Suffix array:** holds starting position of each suffix beginning a row.

panamabanan**as$**

13	**$**$_1$panamabananas$_1$
5	**a**$_1$**bananas$**panam$_1$
3	**a**$_2$**mabananas$**pan$_1$
1	**a**$_3$**namabananas$**p$_1$
7	**a**$_4$**nanas$**panamab$_1$
9	**a**$_5$**nas$**panamaban$_2$
11	**a**$_6$**s$**panamabanan$_3$
	b$_1$ananas$panama$_1$
	m$_1$abananas$pana$_2$
	n$_1$amabananaspa_3$
	n$_2$anas$panamaba$_4$
	n$_3$as$panamabana$_5$
	p$_1$anamabananas$$_1$
	s$_1$$panamabanana$_6$

233

Using the Suffix Array to Find Matches

- **Suffix array:** holds starting position of each suffix beginning a row.

panama**bananas$**

13	**$**$_1$panamabananas$_1$
5	**a**$_1$**bananas$**panam$_1$
3	**a**$_2$**mabananas$**pan$_1$
1	**a**$_3$**namabananas$**p$_1$
7	**a**$_4$**nanas$**panamab$_1$
9	**a**$_5$**nas$**panamaban$_2$
11	**a**$_6$**s$**panamabanan$_3$
6	**b**$_1$**ananas$**panama$_1$
	m$_1$abananas$pana$_2$
	n$_1$amabananaspa_3$
	n$_2$anas$panamaba$_4$
	n$_3$as$panamabana$_5$
	p$_1$anamabananas$$_1$
	s$_1$$panamabanana$_6$

234

Using the Suffix Array to Find Matches

- **Suffix array:** holds starting position of each suffix beginning a row.

pana**mabananas$**

13	**$**$_1$panamabananas$_1$
5	**a**$_1$**bananas$**panam$_1$
3	**a**$_2$**mabananas$**pan$_1$
1	**a**$_3$**namabananas$**p$_1$
7	**a**$_4$**nanas$**panamab$_1$
9	**a**$_5$**nas$**panamaban$_2$
11	**a**$_6$**s$**panamabanan$_3$
6	**b**$_1$**ananas$**panama$_1$
4	**m**$_1$**abananas$**pana$_2$
	n$_1$amabananaspa_3$
	n$_2$anas$panamaba$_4$
	n$_3$as$panamabana$_5$
	p$_1$anamabananas$_1$
	s$_1$$panamabanana$_6$

235

Using the Suffix Array to Find Matches

- **Suffix array:** holds starting position of each suffix beginning a row.

pa**namabananas$**

13	**$**$_1$panamabananas$_1$
5	**a**$_1$**bananas$**panam$_1$
3	**a**$_2$**mabananas$**pan$_1$
1	**a**$_3$**namabananas$**p$_1$
7	**a**$_4$**nanas$**panamab$_1$
9	**a**$_5$**nas$**panamaban$_2$
11	**a**$_6$**s$**panamabanan$_3$
6	**b**$_1$**ananas$**panama$_1$
4	**m**$_1$**abananas$**pana$_2$
2	**n**$_1$**amabananas$**pa$_3$
	n$_2$anas$panamaba$_4$
	n$_3$as$panamabana$_5$
	p$_1$anamabananas$_1$
	s$_1$$panamabanana$_6$

236

Using the Suffix Array to Find Matches

- **Suffix array:** holds starting position of each suffix beginning a row.

panamaba**nanas$**

13	**$**$_1$panamabananas$_1$
5	**a**$_1$**bananas$**panam$_1$
3	**a**$_2$**mabananas$**pan$_1$
1	**a**$_3$**namabananas$**p$_1$
7	**a**$_4$**nanas$**panamab$_1$
9	**a**$_5$**nas$**panamaban$_2$
11	**a**$_6$**s$**panamabanan$_3$
6	**b**$_1$**ananas$**panama$_1$
4	**m**$_1$**abananas$**pana$_2$
2	**n**$_1$**amabananas$**pa$_3$
8	**n**$_2$**anas$**panamaba$_4$
	n$_3$as$panamabana$_5$
	p$_1$anamabananas$$_1$
	s$_1$$panamabanana$_6$

237

Using the Suffix Array to Find Matches

- **Suffix array:** holds starting position of each suffix beginning a row.

panamabana**nas$**

13	**$**$_1$panamabananas$_1$
5	**a**$_1$**bananas$**panam$_1$
3	**a**$_2$**mabananas$**pan$_1$
1	**a**$_3$**namabananas$**p$_1$
7	**a**$_4$**nanas$**panamab$_1$
9	**a**$_5$**nas$**panamaban$_2$
11	**a**$_6$**s$**panamabanan$_3$
6	**b**$_1$**ananas$**panama$_1$
4	**m**$_1$**abananas$**pana$_2$
2	**n**$_1$**amabananas$**pa$_3$
8	**n**$_2$**anas$**panamaba$_4$
10	**n**$_3$**as$**panamabana$_5$
	p$_1$anamabananas$$_1$
	s$_1$$panamabanana$_6$

238

Using the Suffix Array to Find Matches

- **Suffix array:** holds starting position of each suffix beginning a row.

panamabananas$

13	$1panamabananas1
5	a1bananas$panam1
3	a2mabananas$pan1
1	a3namabananas$p1
7	a4nanas$panamab1
9	a5nas$panamaban2
11	a6s$panamabanan3
6	b1ananas$panama1
4	m1abananas$pana2
2	n1amabananas$pa3
8	n2anas$panamaba4
10	n3as$panamabana5
0	p1anamabananas$1
	s1$panamabanana6

239

Using the Suffix Array to Find Matches

- **Suffix array:** holds starting position of each suffix beginning a row.

panamabanana**s$**

13	$1panamabananas1
5	a1bananas$panam1
3	a2mabananas$pan1
1	a3namabananas$p1
7	a4nanas$panamab1
9	a5nas$panamaban2
11	a6s$panamabanan3
6	b1ananas$panama1
4	m1abananas$pana2
2	n1amabananas$pa3
8	n2anas$panamaba4
10	n3as$panamabana5
0	p1anamabananas$1
12	s1$panamabanana6

240

Using the Suffix Array to Find Matches

- **Suffix array:** holds starting position of each suffix beginning a row.

panamabananas$

13	$1panamabananas1
5	a1bananas$panam1
3	a2mabananas$pan1
1	a3namabananas$p1
7	a4nanas$panamab1
9	a5nas$panamaban2
11	a6s$panamabanan3
6	b1ananas$panama1
4	m1abananas$pana2
2	n1amabananas$pa3
8	n2anas$panamaba4
10	n3as$panamabana5
0	p1anamabananas$1
12	s1$panamabanana6

241

Using the Suffix Array to Find Matches

- **Suffix array:** holds starting position of each suffix beginning a row.
- Thus, **ana** occurs at positions **1**, **7**, **9** of p**ana**mab**anana**s$.

13	$1panamabananas1
5	a1bananas$panam1
3	a2mabananas$pan1
1	**a3na**mabananas$p1
7	**a4na**nas$panamab1
9	**a5na**s$panamaban2
11	a6s$panamabanan3
6	b1ananas$panama1
4	m1abananas$pana2
2	n1amabananas$pa3
8	n2anas$panamaba4
10	n3as$panamabana5
0	p1anamabananas$1
12	s1$panamabanana6

242

The Suffix Array: Memory Once Again

- Memory: ~ 4 x |*Genome*|.

time complexity later…

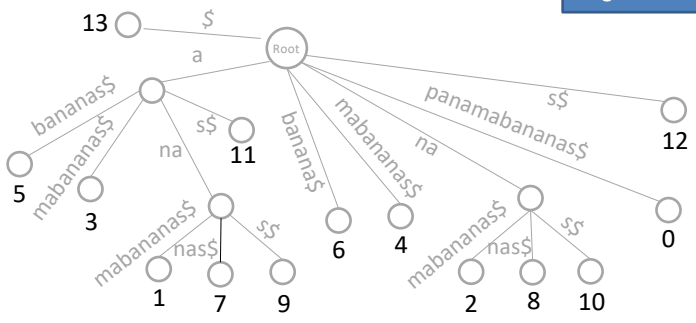
243

The Suffix Array: Memory Once Again

From ST to SA

ST hides the SA (in-order visit if the edges are lex. ordered)

- Memory: ~ 4 x |*Genome*|.



[13 5 3 1 7 9 11 6 4 2 8 10 0 12]

244

From SA to BWT

```
s = appellee$
    123456789
```

9
1
8
7
4
6
5
3
2

Suffix array
(start position
for the suffixes)

245

From SA to BWT

```
s = appellee$
    123456789
```

9
1
8
7
4
6
5
3
2

– subtract 1 →

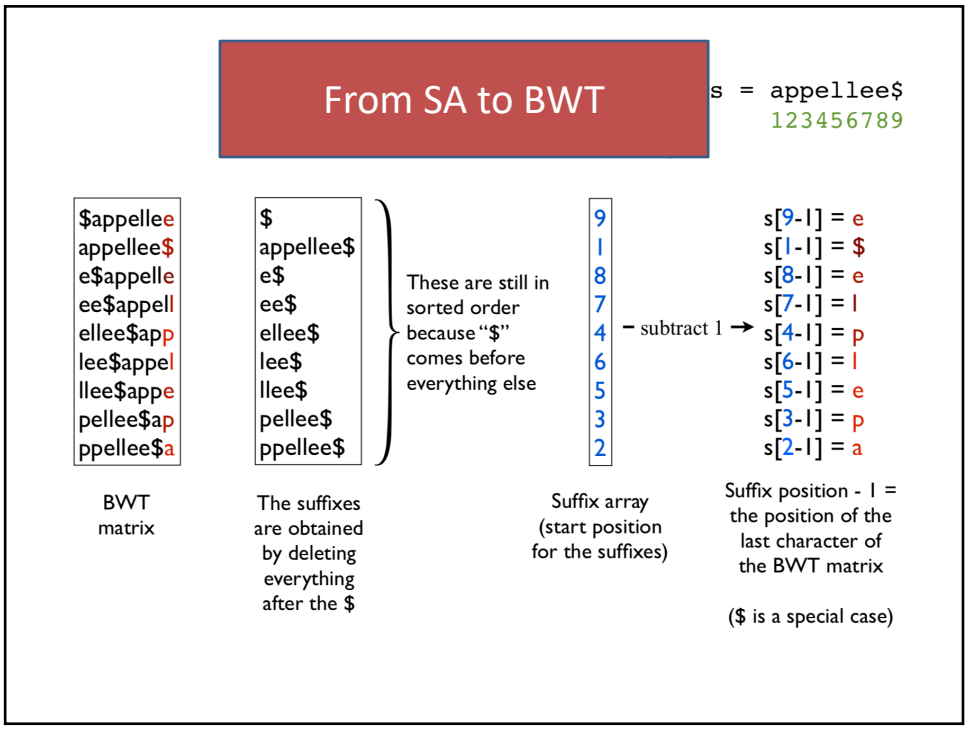
s[9-1] = e
s[1-1] = $
s[8-1] = e
s[7-1] = l
s[4-1] = p
s[6-1] = l
s[5-1] = e
s[3-1] = p
s[2-1] = a

Suffix array
(start position
for the suffixes)

Suffix position - 1 =
the position of the
last character of
the BWT matrix

($ is a special case)

246



247

Outline

- Why do we map reads?
- Using the Trie
- From a Trie to a Suffix Tree
- String Compression and the Burrows-Wheeler Transform
- Inverting Burrows-Wheeler
- Using Burrows-Wheeler for Pattern Matching
- Finding the Matched Patterns
- **Setting Up Checkpoints**
- Inexact Matching
- Further Applications of Read Mapping

248

Reducing Suffix Array Size

- We don’t want to have to store all of the suffix array; can we store only part of it? Show how checkpointing can be used to store 1/100 the suffix array.

chi vuole approfondire, può seguire dal testo/youtube (progetto)

249

Outline

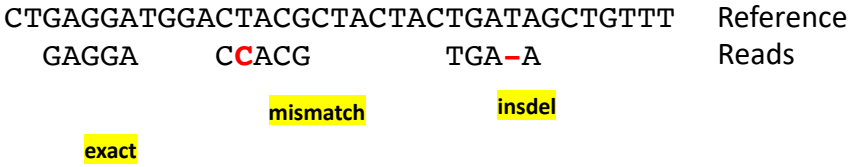
- Why do we map reads?
- Using the Trie
- From a Trie to a Suffix Tree
- String Compression and the Burrows-Wheeler Transform
- Inverting Burrows-Wheeler
- Using Burrows-Wheeler for Pattern Matching
- Finding the Matched Patterns
- Setting Up Checkpoints
- **Inexact Matching**
- Further Applications of Read Mapping

250

Read Mapping

- **Read mapping**: determine where each read has high similarity to the reference genome.

without assemblying



251

Returning to Our Original Problem

- We need to look at INEXACT matching in order to find variants.
- **Approx. Pattern Matching Problem**:
  - **Input**: A string *Pattern*, a string *Genome*, and an integer *d*.
  - **Output:** All positions in *Genome* where *Pattern* appears as a substring with at most *d* mismatches.

252

Returning to Our Original Problem

- We need to look at INEXACT matching in order to find variants.
- **Multiple Approx. Pattern Matching Problem**:
  - **Input**: A **collection** of strings *Patterns*, a string *Genome*, and an integer *d*.
  - **Output:** All positions in *Genome* where a string from *Patterns* appears as a substring with at most *d* mismatches.

253

Method 1: Seeding

- Say we are looking for at most *d* mismatches.
- Divide each of our strings into *d* + 1 smaller pieces, called **seeds**.
- Check if each *Pattern* has a seed that matches *Genome* exactly.
- If so, check the entire *Pattern* against *Genome*.

254

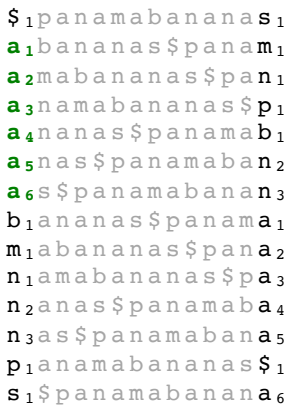
Method 2: BWT Saves the Day Again

- Recall: searching for ana in panamabananas

255

Method 2: BWT Saves the Day Again

- Recall: searching for ana in panamabananas



256

Method 2: BWT Saves the Day Again

- Recall: searching for **ana** in panamabananas

$_1$panamabananas$_1$
a$_1$bananas$panam**m**$_1$
a$_2$mabananas$pan**n**$_1$
a$_3$namabananas$**p**$_1$
a$_4$nanas$panama**b**$_1$
a$_5$nas$panamaba**n**$_2$
a$_6$s$panamabana**n**$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamaban$a_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$

257

Method 2: BWT Saves the Day Again

- Recall: searching for **ana** in panamabananas

If we allow 1 mismatch, then we need to keep the red letters around.

$_1$panamabananas$_1$
a$_1$bananas$panam**m**$_1$
a$_2$mabananas$pan**n**$_1$
a$_3$namabananas$**p**$_1$
a$_4$nanas$panama**b**$_1$
a$_5$nas$panamaba**n**$_2$
a$_6$s$panamabana**n**$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamaba$_4$
n$_3$as$panamaban$a_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanana$_6$

258

Method 2: BWT Saves the Day Again

- Recall: searching for **ana** in panamabananas

If we allow 1 mismatch, then we need to keep the red letters around.

	# Mismatches
$$_1$panamabananas$_1$	
a$_1$bananas$panam**m**$_1$	
a$_2$mabananas$pan**n**$_1$	
a$_3$namabananas$**p**$_1$	
a$_4$nanas$panama**b**$_1$	
a$_5$nas$panamaba**n**$_2$	
a$_6$s$panamaban**n**$_3$	
b$_1$ananas$panama$_1$	
m$_1$abananas$pana$_2$	
n$_1$amabananaspa_3$	
n$_2$anas$panamaba$_4$	
n$_3$as$panamaban$_5$	
p$_1$anamabananas$$_1$	
s$_1$$panamabanana$_6$	

259

Method 2: BWT Saves the Day Again

- Recall: searching for **ana** in panamabananas

If we allow 1 mismatch, then we need to keep the red letters around.

	# Mismatches
$$_1$panamabananas$_1$	
a$_1$bananas$panam**m**$_1$	1
a$_2$mabananas$pan**n**$_1$	0
a$_3$namabananas$**p**$_1$	1
a$_4$nanas$panama**b**$_1$	1
a$_5$nas$panamaba**n**$_2$	0
a$_6$s$panamaban**n**$_3$	0
b$_1$ananas$panama$_1$	
m$_1$abananas$pana$_2$	
n$_1$amabananaspa_3$	
n$_2$anas$panamaba$_4$	
n$_3$as$panamaban$_5$	
p$_1$anamabananas$$_1$	
s$_1$$panamabanana$_6$	

260

Method 2: BWT Saves the Day Again

- Recall: searching for **ana** in panamabananas

Now we extend all strings with at most 1 mismatch.

	# Mismatches
$$_1$panamabananas$_1$	
a$_1$bananas$panam$_1$	1
a$_2$mabananaspan_1$	0
a$_3$namabananas$p$_1$	1
a$_4$nanas$panamab$_1$	1
a$_5$nas$panamaban$_2$	0
a$_6$s$panamabanan$_3$	0
b$_1$ananas$panama$_1$	
m$_1$abananas$pana$_2$	
n$_1$amabananaspa_3$	
n$_2$anas$panamaba$_4$	
n$_3$as$panamaban$_5$	
p$_1$anamabananas$$_1$	
s$_1$$panamabanana$_6$	

261

Method 2: BWT Saves the Day Again

- Recall: searching for **ana** in panamabananas

Now we extend all strings with at most 1 mismatch.

	# Mismatches
$$_1$panamabananas$_1$	
a$_1$bananas$panam$_1$	1
a$_2$mabananaspan_1$	0
a$_3$namabananas$p$_1$	1
a$_4$nanas$panamab$_1$	1
a$_5$nas$panamaban$_2$	0
a$_6$s$panamabanan$_3$	0
b$_1$ananas$panama$_1$	
m$_1$abananas$pana$_2$	
n$_1$amabananaspa_3$	
n$_2$anas$panamaba$_4$	
n$_3$as$panamaban$_5$	
p$_1$anamabananas$$_1$	
s$_1$$panamabanana$_6$	

262

Method 2: BWT Saves the Day Again

- Recall: searching for **ana** in panamabananas

Now we extend all strings with at most 1 mismatch.

	# Mismatches
$$_1$panamabananas$_1$	
a$_1$bananas$$panam**m**$_1$	1
a$_2$mabananas$$pan**n**$_1$	0
a$_3$namabananas$$**p**$_1$	1
a$_4$nanas$$panama**b**$_1$	1
a$_5$nas$$panamaba**n**$_2$	0
a$_6$s$$panamaban**n**$_3$	0
b$_1$**a**nanas$$panama$_1$	
m$_1$**a**bananas$$pana$_2$	
n$_1$**a**mabananas$$pa$_3$	
n$_2$**a**nas$$panamaba$_4$	
n$_3$**a**s$$panamaban**a**$_5$	
p$_1$**a**namabananas$$_1$	
s$_1$$$panamabanan**a**$_6$	

263

Method 2: BWT Saves the Day Again

- Recall: searching for **ana** in panamabananas

Now we extend all strings with at most 1 mismatch.

	# Mismatches
$$_1$panamabananas$_1$	
a$_1$bananas$$panam$_1$	
a$_2$mabananas$$pan$_1$	
a$_3$namabananas$$p$_1$	
a$_4$nanas$$panamab$_1$	
a$_5$nas$$panamaban$_2$	
a$_6$s$$panamabanan$_3$	
b$_1$**a**nanas$$panama$_1$	1
m$_1$**a**bananas$$pana$_2$	1
n$_1$**a**mabananas$$pa$_3$	0
n$_2$**a**nas$$panamaba$_4$	0
n$_3$**a**s$$panamaban**a**$_5$	0
p$_1$**a**namabananas$$_1$	1
s$_1$$$panamabanan**a**$_6$	

264

Method 2: BWT Saves the Day Again

- Recall: searching for **ana** in panamabananas

One string produces a second mismatch (the $), so we discard it.

	# Mismatches
$$_1$panamabananas$_1$	
a$_1$bananas$panam$_1$	
a$_2$mabananaspan_1$	
a$_3$namabananas$p$_1$	
a$_4$nanas$panamab$_1$	
a$_5$nas$panamaban$_2$	
a$_6$s$panamabanan$_3$	
b$_1$ananas$panam**a$_1$**	1
m$_1$abananas$pan**a$_2$**	1
n$_1$amabananas$p**a$_3$**	0
n$_2$anas$panamab**a$_4$**	0
n$_3$as$panamaban**a$_5$**	0
p$_1$anamabananas**$$_1$**	2
s$_1$$panamabanan**a$_6$**	

265

Method 2: BWT Saves the Day Again

- Recall: searching for **ana** in panamabananas

In the end, we have five 3-mers with at most 1 mismatch.

	# Mismatches
$$_1$panamabananas$_1$	
a$_1$bananas$panam$_1$	1
a$_2$mabananaspan_1$	1
a$_3$namabananas$p$_1$	0
a$_4$nanas$panamab$_1$	0
a$_5$nas$panamaban$_2$	0
a$_6$s$panamabanan$_3$	
b$_1$ananas$panam**a$_1$**	
m$_1$abananas$pan**a$_2$**	
n$_1$amabananas$p**a$_3$**	
n$_2$anas$panamab**a$_4$**	
n$_3$as$panamaban**a$_5$**	
p$_1$anamabananas**$$_1$**	
s$_1$$panamabanana_6	

266

Method 2: BWT Saves the Day Again

- Recall: searching for **ana** in panamabananas

In the end, we have five 3-mers with at most 1 mismatch.

	# Mismatches
$$_1$panamabananas$_1$	
a$_1$**ba**nanas$panam$_1$	1
a$_2$**ma**bananaspan_1$	1
a$_3$**na**mabananas$p$$_1$	0
a$_4$**na**nas$panamab$$_1$	0
a$_5$**na**s$panamaban$$_2$	0
a$_6$s$panamabanan$$_3$	
b$_1$ananas$panama$$_1$	
m$_1$abananas$pana$$_2$	
n$_1$amabananas$pa$$_3$	
n$_2$anas$panamaba$$_4$	
n$_3$as$panamabana$$_5$	
p$_1$anamabananas$$_1$	
s$_1$$panamabanana$$_6$	

267

Method 2: BWT Saves the Day Again

- Recall: searching for **ana** in panamabananas

In the end, we have five 3-mers with at most 1 mismatch.

	Suffix Array
$$_1$panamabananas$_1$	
a$_1$**ba**nanas$panam$_1$	5
a$_2$**ma**bananaspan_1$	3
a$_3$**na**mabananas$p$$_1$	1
a$_4$**na**nas$panamab$$_1$	7
a$_5$**na**s$panamaban$$_2$	9
a$_6$s$panamabanan$$_3$	
b$_1$ananas$panama$$_1$	
m$_1$abananas$pana$$_2$	
n$_1$amabananas$pa$$_3$	
n$_2$anas$panamaba$$_4$	
n$_3$as$panamabana$$_5$	
p$_1$anamabananas$$_1$	
s$_1$$panamabanana$$_6$	

268

Method 2: BWT Saves the Day Again

- Recall: searching for **ana** in panamabananas

In the end, we have five 3-mers with at most 1 mismatch.

	Suffix Array
$$_1$panamabananas$_1$	
a$_1$**ba**nanas$panam$_1$	**5**
a$_2$mabananaspan_1$	3
a$_3$namabananas$p$_1$	1
a$_4$nanas$panamab$_1$	7
a$_5$nas$panamaban$_2$	9
a$_6$s$panamabanan$_3$	
b$_1$ananas$panama$_1$	
m$_1$abananas$pana$_2$	
n$_1$amabananaspa_3$	
n$_2$anas$panamaba$_4$	
n$_3$as$panamabana$_5$	
p$_1$anamabananas$$_1$	
s$_1$$panamabanana$_6$	

269

Method 2: BWT Saves the Day Again

- Recall: searching for **ana** in panamabananas

In the end, we have five 3-mers with at most 1 mismatch.

	Suffix Array
$$_1$panamabananas$_1$	
a$_1$bananas$panam$_1$	5
a$_2$**ma**bananaspan_1$	**3**
a$_3$namabananas$p$_1$	1
a$_4$nanas$panamab$_1$	7
a$_5$nas$panamaban$_2$	9
a$_6$s$panamabanan$_3$	
b$_1$ananas$panama$_1$	
m$_1$abananas$pana$_2$	
n$_1$amabananaspa_3$	
n$_2$anas$panamaba$_4$	
n$_3$as$panamabana$_5$	
p$_1$anamabananas$$_1$	
s$_1$$panamabanana$_6$	

270

Method 2: BWT Saves the Day Again

- Recall: searching for **ana** in panamabananas

In the end, we have five 3-mers with at most 1 mismatch.

	Suffix Array
$1panamabananas1	
a1bananas$panam1	5
a2mabananas$pan1	3
a3namabananas$p1	**1**
a4nanas$panamab1	7
a5nas$panamaban2	9
a6s$panamabanan3	
b1ananas$panama1	
m1abananas$pana2	
n1amabananas$pa3	
n2anas$panamaba4	
n3as$panamabana5	
p1anamabananas$1	
s1$panamabanana6	

271

Method 2: BWT Saves the Day Again

- Recall: searching for **ana** in panamab**ana**nas

In the end, we have five 3-mers with at most 1 mismatch.

	Suffix Array
$1panamabananas1	
a1bananas$panam1	5
a2mabananas$pan1	3
a3namabananas$p1	1
a4nanas$panamab1	**7**
a5nas$panamaban2	9
a6s$panamabanan3	
b1ananas$panama1	
m1abananas$pana2	
n1amabananas$pa3	
n2anas$panamaba4	
n3as$panamabana5	
p1anamabananas$1	
s1$panamabanana6	

272

Method 2: BWT Saves the Day Again

- Recall: searching for **ana** in panamabanan**as**

In the end, we have five 3-mers with at most 1 mismatch.

	Suffix Array
$$_1$panamabananas$_1$	
a$_1$bananas$$_1$panam$_1$	5
a$_2$mabananas$$_1pan_1$	3
a$_3$namabananas$$_1p_1$	1
a$_4$nanas$$_1$panamab$_1$	7
a$_5$nas$$_1$panamaban$_2$	**9**
a$_6$s$$_1$panamabanan$_3$	
b$_1$ananas$$_1$panama$_1$	
m$_1$abananas$$_1$pana$_2$	
n$_1$amabananas$$_1pa_3$	
n$_2$anas$$_1$panamaba$_4$	
n$_3$as$$_1$panamaban$_5$	
p$_1$anamabananas$$_1$	
s$_1$$$_1$panamabanana$_6$	

273

Citations

Compression

[PDF] A block-sorting lossless data compression algorithm
www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf
by AB Lossless - 1994 - Related articles
May 10, 1994 - **A Block-sorting Lossless**. **Data Compression Algorithm**. M. Burrows and D.J. Wheeler digital. Systems Research Center. 130 Lytton Avenue.

Pattern Matching

Opportunistic data structures with applications - IEEE Conference ...
https://ieeexplore.ieee.org/document/892127
by P Ferragina - 2000 - Cited by 1016 - Related articles
Opportunistic data structures with applications. Abstract: We address the issue of compressing and **indexing** data. We devise a data structure whose space occupancy is a function of the entropy of the underlying data set.

Read Mapping

Fast and accurate short read alignment with Burrows–Wheeler transform
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2705234/
by H Li - 2009 - Cited by 17996 - Related articles
May 18, 2009 - Results: We implemented Burrows-Wheeler Alignment tool (**BWA**), a new read alignment package that is based on backward search with ...

Ultrafast and memory-efficient alignment of short DNA sequences to ...
https://genomebiology.biomedcentral.com/articles/10.1186/gb-2009-10-3-r25
by B Langmead - 2009 - Cited by 13797 - Related articles
Mar 4, 2009 - **Bowtie** is an ultrafast, memory-efficient alignment program for aligning short DNA sequence reads to large genomes. For the human genome, ...

274

From BWT + FM to SA

The FM-index

- It is a 2D array whose size is $[|text| + 1, |\Sigma|]$, where $|\Sigma|$ is the alphabet size
- It keeps track of how many of each symbol have been seen in the BWT prior to its i^{th} symbol
- The last m row is the totals for each symbol. By accumulating these totals you can determine the BWT index corresponding to the first of each symbol in the suffix array (Offset).
- Can be generated by a single scan through the BWT
- Memory overhead $O(m|\Sigma|)$

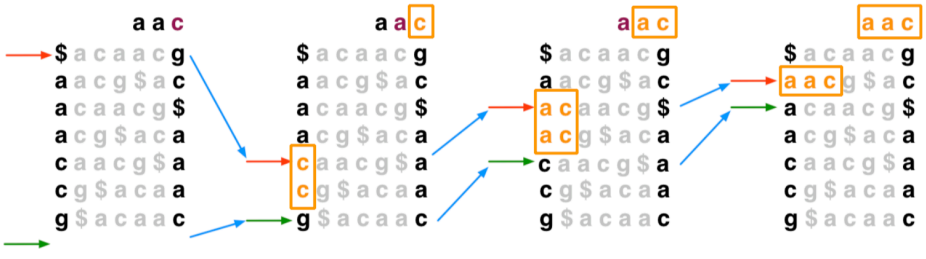
			FM-index			
Index	Suffix Array	BWT	$	a	b	n
0	$banana	a	0	0	0	0
1	a$banan	n	0	1	0	0
2	ana$ban	n	0	1	0	1
3	anana$b	b	0	1	0	2
4	banana$	$	0	1	1	2
5	na$bana	a	1	1	1	2
6	nana$ba	a	1	2	1	2
7	**Counts**		1	3	1	2
	Offset		0	1	4	5

Counts(x)=numero di occorrenze di x
Occ(x,i)=numero di caratteri x in BWT[0,i-1], i>0.
Offset: accumulo i conteggi per scoprire la prima occorrenza nel SA
$ è in 0
a è in 0+1=1
b è in 1+3=4
n è in 4+1=5

275

BWT+FM: miglioro la ricerca binaria già vista…

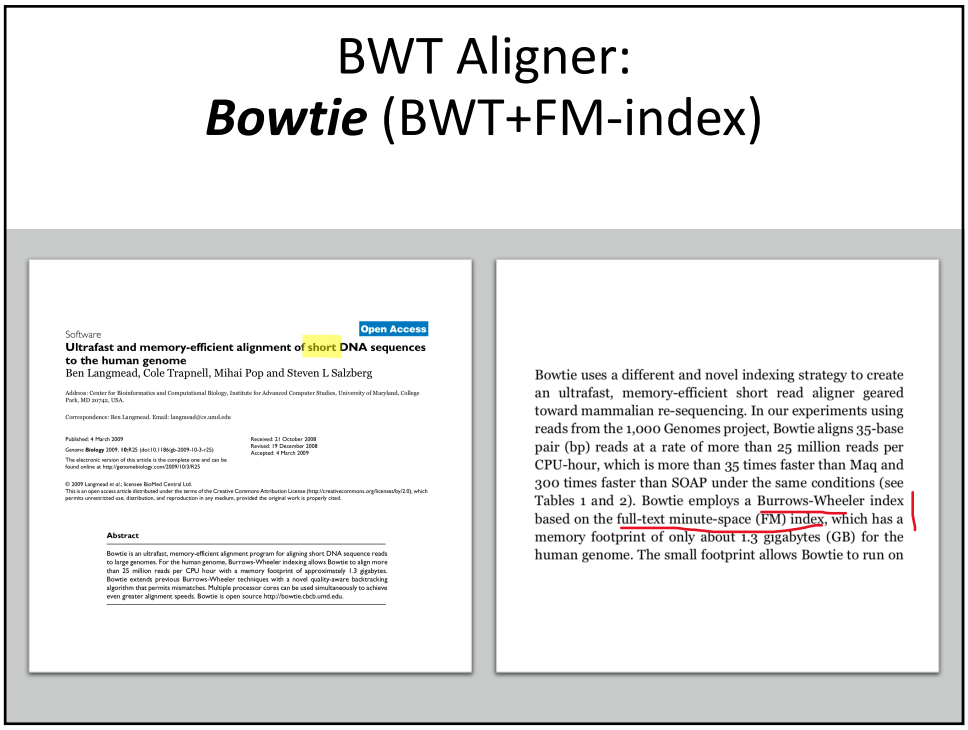
Exact Matching with FM Index



- In progressive rounds, **top** & **bot** delimit the range of rows beginning with progressively longer suffixes of Q

276

BWT Aligner: ***Bowtie*** (BWT+FM-index)



Software

Open Access

Ultrafast and memory-efficient alignment of short DNA sequences to the human genome

Ben Langmead, Cole Trapnell, Mihai Pop and Steven L Salzberg

Address: Center for Bioinformatics and Computational Biology, Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA.

Correspondence: Ben Langmead. Email: langmead@cs.umd.edu

Published: 4 March 2009

Genome Biology 2009, 10:R25 (doi:10.1186/gb-2009-10-3-r25)

The electronic version of this article is the complete one and can be found online at http://genomebiology.com/2009/10/3/R25

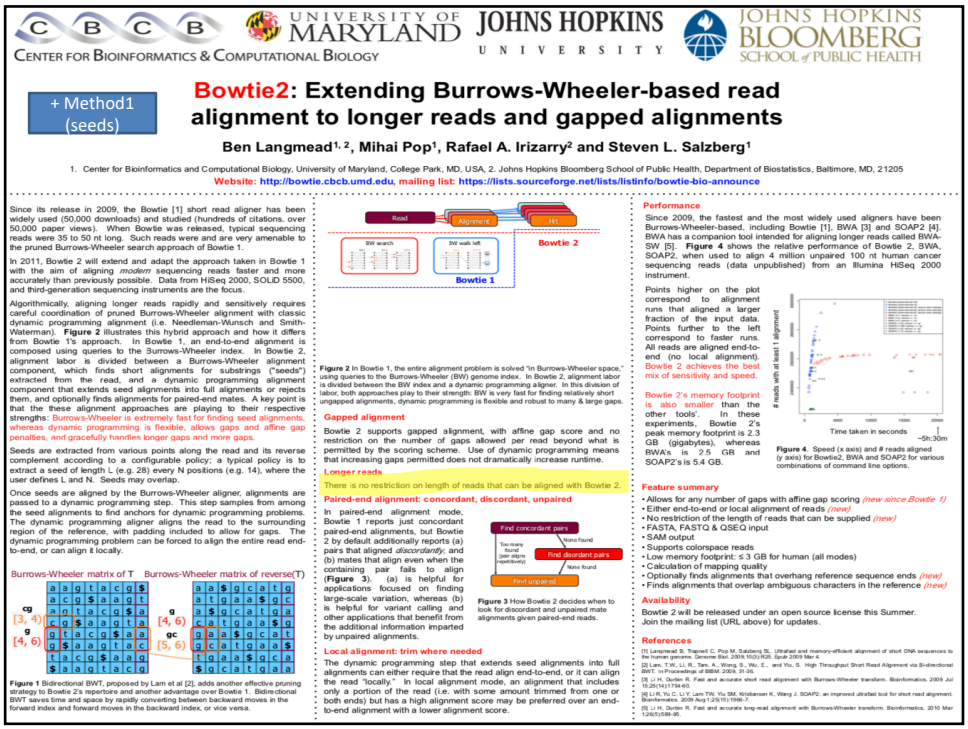
Received: 31 October 2008
Revised: 19 December 2008
Accepted: 4 March 2009

Abstract

Bowtie is an ultrafast, memory-efficient alignment program for aligning short DNA sequence reads to large genomes. For the human genome, Burrows-Wheeler indexing allows Bowtie to align more than 25 million reads per CPU hour with a memory footprint of approximately 1.3 gigabytes. Bowtie extends previous Burrows-Wheeler techniques with a novel quality-aware backtracking algorithm that permits mismatches. Multiple processor cores can be used simultaneously to achieve even greater alignment speeds. Bowtie is open source http://bowtie.cbcb.umd.edu.

Bowtie uses a different and novel indexing strategy to create an ultrafast, memory-efficient short read aligner geared toward mammalian re-sequencing. In our experiments using reads from the 1,000 Genomes project, Bowtie aligns 35-base pair (bp) reads at a rate of more than 25 million reads per CPU-hour, which is more than 35 times faster than Maq and 300 times faster than SOAP under the same conditions (see Tables 1 and 2). Bowtie employs a Burrows-Wheeler index based on the full-text minute-space (FM) index, which has a memory footprint of only about 1.3 gigabytes (GB) for the human genome. The small footprint allows Bowtie to run on

277



CBCB CENTER FOR BIOINFORMATICS & COMPUTATIONAL BIOLOGY — UNIVERSITY OF MARYLAND — JOHNS HOPKINS UNIVERSITY — JOHNS HOPKINS BLOOMBERG SCHOOL of PUBLIC HEALTH

+ Method1 (seeds)

Bowtie2: Extending Burrows-Wheeler-based read alignment to longer reads and gapped alignments

Ben Langmead[1, 2], Mihai Pop[1], Rafael A. Irizarry[2] and Steven L. Salzberg[1]

1. Center for Bioinformatics and Computational Biology, University of Maryland, College Park, MD, USA. 2. Johns Hopkins Bloomberg School of Public Health, Department of Biostatistics, Baltimore, MD, 21205

Website: http://bowtie.cbcb.umd.edu, mailing list: https://lists.sourceforge.net/lists/listinfo/bowtie-bio-announce

Since its release in 2009, the Bowtie [1] short read aligner has been widely used (50,000 downloads) and studied (hundreds of citations, over 50,000 paper views). When Bowtie was released, typical sequencing reads were 35 to 50 nt long. Such reads were and are very amenable to the pruned Burrows-Wheeler search approach of Bowtie 1.

In 2011, Bowtie 2 will extend and adapt the approach taken in Bowtie 1 with the aim of aligning *modern* sequencing reads faster and more accurately than previously possible. Data from HiSeq 2000, SOLiD 5500, and third-generation sequencing instruments are the focus.

Algorithmically, aligning longer reads rapidly and sensitively requires careful coordination of pruned Burrows-Wheeler alignment with classic dynamic programming alignment (i.e. Needleman-Wunsch and Smith-Waterman). **Figure 2** illustrates this hybrid approach and how it differs from Bowtie 1's approach. In Bowtie 1, an end-to-end alignment is composed using queries to the Burrows-Wheeler index. In Bowtie 2, alignment labor is divided between a Burrows-Wheeler alignment component, which finds short alignments for substrings ("seeds") extracted from the read, and a dynamic programming alignment component that extends seed alignments into full alignments or rejects them, and optionally finds alignments for paired-end mates. A key point is that the these alignment approaches are playing to their respective strengths: Burrows-Wheeler is extremely fast for finding seed alignments, whereas dynamic programming is flexible, allows gaps and affine gap penalties, and gracefully handles longer gaps and more gaps.

Seeds are extracted from various points along the read and its reverse complement according to a configurable policy; a typical policy is to extract a seed of length L (e.g. 28) every N positions (e.g. 14), where the user defines L and N. Seeds may overlap.

Once seeds are aligned by the Burrows-Wheeler aligner, alignments are passed to a dynamic programming step. This step samples from among the seed alignments to find anchors for dynamic programming problems. The dynamic programming aligner aligns the read to the surrounding region of the reference, with padding included to allow for gaps. The dynamic programming problem can be forced to align the entire read end-to-end, or can align it locally.

Burrows-Wheeler matrix of T Burrows-Wheeler matrix of reverse(T)

Figure 1 Bidirectional BWT, proposed by Lam et al [2], adds another effective pruning strategy to Bowtie 2's repertoire and another advantage over Bowtie 1. Bidirectional BWT saves time and space by rapidly converting between backward moves in the forward index and forward moves in the backward index, or vice versa.

Figure 2 In Bowtie 1, the entire alignment problem is solved "in Burrows-Wheeler space," using queries to the Burrows-Wheeler (BW) genome index. In Bowtie 2, alignment labor is divided between the BW index and a dynamic programming aligner. In this division of labor, both approaches play to their strength: BW is very fast for finding relatively short ungapped alignments, dynamic programming is flexible and robust to many & large gaps.

Gapped alignment

Bowtie 2 supports gapped alignment, with affine gap score and no restriction on the number of gaps allowed per read beyond what is permitted by the scoring scheme. Use of dynamic programming means that increasing gaps permitted does not dramatically increase runtime.

Longer reads

There is no restriction on length of reads that can be aligned with Bowtie 2.

Paired-end alignment: concordant, discordant, unpaired

In paired-end alignment mode, Bowtie 1 reports just concordant paired-end alignments, but Bowtie 2 by default additionally reports (a) pairs that aligned *discordantly*; and (b) mates that align even when the containing pair fails to align (**Figure 3**). (a) is helpful for applications focused on finding large-scale variation, whereas (b) is helpful for variant calling and other applications that benefit from the additional information imparted by unpaired alignments.

Figure 3 How Bowtie 2 decides when to look for discordant and unpaired mate alignments given paired-end reads.

Local alignment: trim where needed

The dynamic programming step that extends seed alignments into full alignments can either require that the read align end-to-end, or it can align the read "locally." In local alignment mode, an alignment that includes only a portion of the read (i.e. with some amount trimmed from one or both ends) but has a high alignment score may be preferred over an end-to-end alignment with a lower alignment score.

Performance

Since 2009, the fastest and the most widely used aligners have been Burrows-Wheeler-based, including Bowtie [1], BWA [3] and SOAP2 [4]. BWA has a companion tool intended for aligning longer reads called BWA-SW [5]. **Figure 4** shows the relative performance of Bowtie 2, BWA, SOAP2, when used to align 4 million unpaired 100 nt human cancer sequencing reads (data unpublished) from an Illumina HiSeq 2000 instrument.

Points higher on the plot correspond to alignment runs that aligned a larger fraction of the input data. Points further to the left correspond to faster runs. All reads are aligned end-to-end (no local alignment). Bowtie 2 achieves the best mix of sensitivity and speed.

Bowtie 2's memory footprint is also smaller than the other tools'. In these experiments, Bowtie 2's peak memory footprint is 2.3 GB (gigabytes), whereas BWA's is 2.5 GB and SOAP2's is 5.4 GB.

Figure 4. Speed (x axis) and # reads aligned (y axis) for Bowtie2, BWA and SOAP2 for various combinations of command line options.

Feature summary

- Allows for any number of gaps with affine gap scoring *(new since Bowtie 1)*
- Either end-to-end or local alignment of reads *(new)*
- No restriction of the length of reads that can be supplied *(new)*
- FASTA, FASTQ & QSEQ input
- SAM output
- Supports colorspace reads
- Low memory footprint: ≤ 3 GB for human (all modes)
- Calculation of mapping quality
- Optionally finds alignments that overhang reference sequence ends *(new)*
- Finds alignments that overlap ambiguous characters in the reference *(new)*

Availability

Bowtie 2 will be released under an open source license this Summer. Join the mailing list (URL above) for updates.

References

[1] Langmead B, Trapnell C, Pop M, Salzberg SL. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol. 2009; 10(3):R25. Epub 2009 Mar 4.

[2] Lam, T.W., Li, R., Tam, A., Wong, S., Wu, E., and Yiu, S. High Throughput Short Read Alignment via Bidirectional BWT. In Proceedings of BIBM 2009, 31-36.

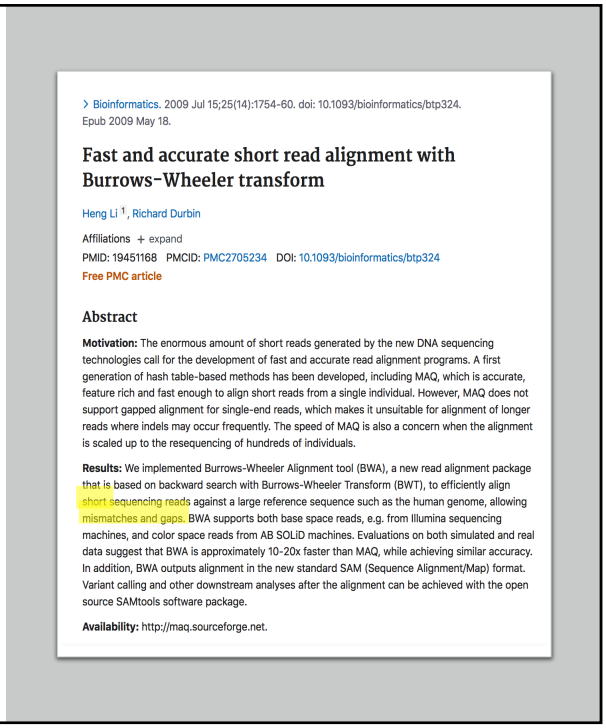
[3] Li H, Durbin R. Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics. 2009 Jul 15;25(14):1754-60.

[4] Li R, Yu C, Li Y, Lam TW, Yiu SM, Kristiansen K, Wang J. SOAP2: an improved ultrafast tool for short read alignment. Bioinformatics. 2009 Aug 1;25(15):1966-7.

[5] Li H, Durbin R. Fast and accurate long-read alignment with Burrows-Wheeler transform. Bioinformatics. 2010 Mar 1;26(5):589-95.

278

BWT Aligner: ***BWA*** (BWT+FM-index…)



> Bioinformatics. 2009 Jul 15;25(14):1754-60. doi: 10.1093/bioinformatics/btp324. Epub 2009 May 18.

Fast and accurate short read alignment with Burrows-Wheeler transform

Heng Li [1], Richard Durbin

Affiliations + expand

PMID: 19451168 PMCID: PMC2705234 DOI: 10.1093/bioinformatics/btp324

Free PMC article

Abstract

Motivation: The enormous amount of short reads generated by the new DNA sequencing technologies call for the development of fast and accurate read alignment programs. A first generation of hash table-based methods has been developed, including MAQ, which is accurate, feature rich and fast enough to align short reads from a single individual. However, MAQ does not support gapped alignment for single-end reads, which makes it unsuitable for alignment of longer reads where indels may occur frequently. The speed of MAQ is also a concern when the alignment is scaled up to the resequencing of hundreds of individuals.

Results: We implemented Burrows-Wheeler Alignment tool (BWA), a new read alignment package that is based on backward search with Burrows-Wheeler Transform (BWT), to efficiently align short sequencing reads against a large reference sequence such as the human genome, allowing mismatches and gaps. BWA supports both base space reads, e.g. from Illumina sequencing machines, and color space reads from AB SOLiD machines. Evaluations on both simulated and real data suggest that BWA is approximately 10-20x faster than MAQ, while achieving similar accuracy. In addition, BWA outputs alignment in the new standard SAM (Sequence Alignment/Map) format. Variant calling and other downstream analyses after the alignment can be achieved with the open source SAMtools software package.

Availability: http://maq.sourceforge.net.

279

BWA vs. Bowtie

Hatem *et al. BMC Bioinformatics* 2013, **14**:184
http://www.biomedcentral.com/1471-2105/14/184

BMC Bioinformatics

RESEARCH ARTICLE Open Access

Benchmarking short sequence mapping tools

Ayat Hatem[1,2], Doruk Bozdağ[2], Amanda E Toland[3] and Ümit V Çatalyürek[1,2*]

Abstract

Background: The development of next-generation sequencing instruments has led to the generation of millions of short sequences in a single run. The process of aligning these reads to a reference genome is time consuming and demands the development of fast and accurate alignment tools. However, the current proposed tools make different compromises between the accuracy and the speed of mapping. Moreover, many important aspects are overlooked while comparing the performance of a newly developed tool to the state of the art. Therefore, there is a need for an objective evaluation method that covers all the aspects. In this work, we introduce a benchmarking suite to extensively analyze sequencing tools with respect to various aspects and provide an objective comparison.

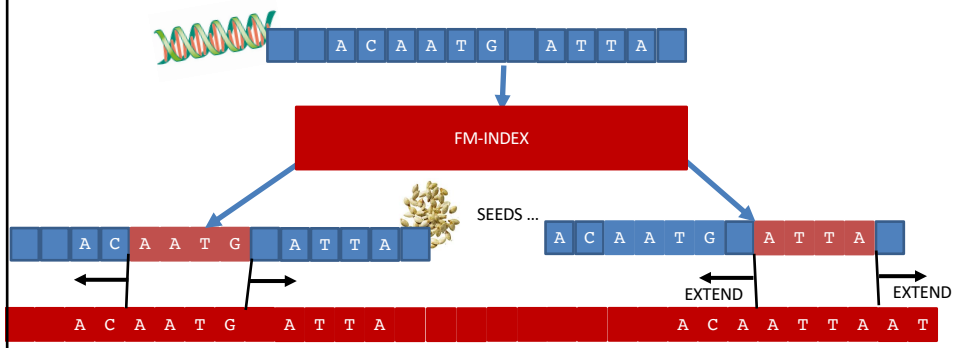
Results: We applied our benchmarking tests on 9 well known mapping tools, namely, Bowtie, Bowtie2, BWA, SOAP2, MAQ, RMAP, GSNAP, Novoalign, and mrsFAST (mrFAST) using synthetic data and real RNA-Seq data. MAQ and RMAP are based on building hash tables for the reads, whereas the remaining tools are based on indexing the reference genome. The benchmarking tests reveal the strengths and weaknesses of each tool. The results show that no single tool outperforms all others in all metrics. However, Bowtie maintained the best throughput for most of the tests while BWA performed better for longer read lengths. The benchmarking tests are not restricted to the mentioned tools and can be further applied to others.

Conclusion: The mapping process is still a hard problem that is affected by many factors. In this work, we provided a benchmarking suite that reveals and evaluates the different factors affecting the mapping process. Still, there is no tool that outperforms all of the others in all the tests. Therefore, the end user should clearly specify his needs in order to choose the tool that provides the best results.

Keywords: Short sequence mapping, Next-generation sequencing, Benchmark, Sequence analysis

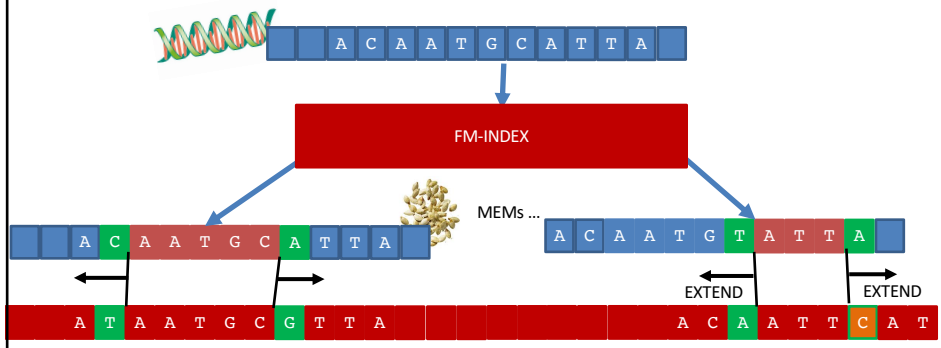
280

Use of FM-index in Read Alignment



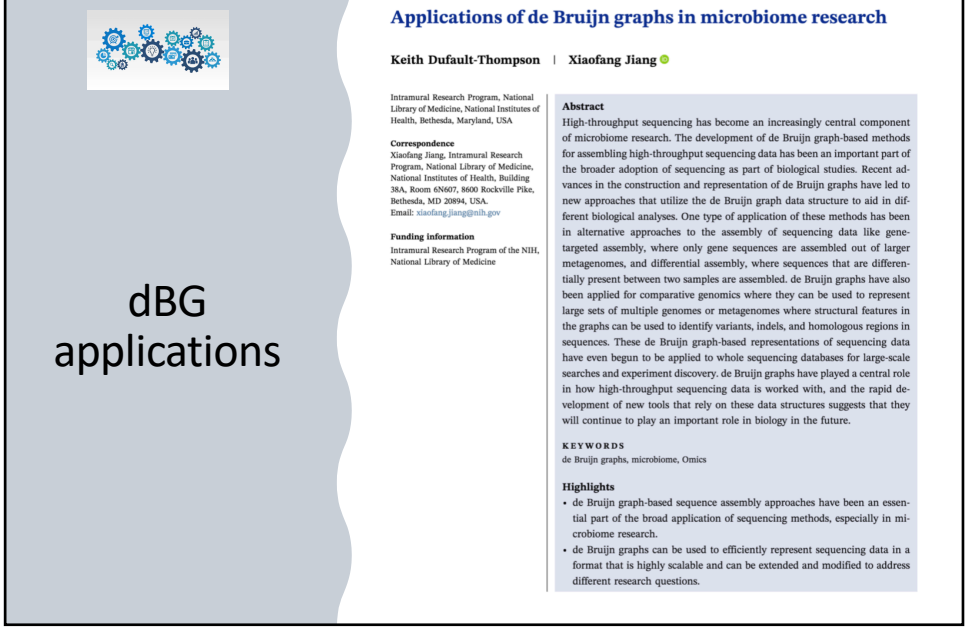
281

MEMs Make Good Seeds



Given a text S[1..n] and a pattern P[1..m]. The substring P [i .. i+ $\ell - 1$] of length ℓ is a maximal exact match (MEM) of P in S if: $P[i \,..\, i+\ell-1]$ occurs in S and $P[i-1..i+\ell-1]$ and $P[i \,..\, i+\ell]$ does not occur in S

282



dBG applications

Received: 20 December 2021 | Revised: 24 January 2022 | Accepted: 24 January 2022
DOI: 10.1002/imt2.4

iMeta WILEY

Applications of de Bruijn graphs in microbiome research

Keith Dufault-Thompson | **Xiaofang Jiang**

Intramural Research Program, National Library of Medicine, National Institutes of Health, Bethesda, Maryland, USA

Correspondence
Xiaofang Jiang, Intramural Research Program, National Library of Medicine, National Institutes of Health, Building 38A, Room 6N607, 8600 Rockville Pike, Bethesda, MD 20894, USA.
Email: xiaofang.jiang@nih.gov

Funding information
Intramural Research Program of the NIH, National Library of Medicine

Abstract

High-throughput sequencing has become an increasingly central component of microbiome research. The development of de Bruijn graph-based methods for assembling high-throughput sequencing data has been an important part of the broader adoption of sequencing as part of biological studies. Recent advances in the construction and representation of de Bruijn graphs have led to new approaches that utilize the de Bruijn graph data structure to aid in different biological analyses. One type of application of these methods has been in alternative approaches to the assembly of sequencing data like gene-targeted assembly, where only gene sequences are assembled out of larger metagenomes, and differential assembly, where sequences that are differentially present between two samples are assembled. de Bruijn graphs have also been applied for comparative genomics where they can be used to represent large sets of multiple genomes or metagenomes where structural features in the graphs can be used to identify variants, indels, and homologous regions in sequences. These de Bruijn graph-based representations of sequencing data have even begun to be applied to whole sequencing databases for large-scale searches and experiment discovery. de Bruijn graphs have played a central role in how high-throughput sequencing data is worked with, and the rapid development of new tools that rely on these data structures suggests that they will continue to play an important role in biology in the future.

KEYWORDS
de Bruijn graphs, microbiome, Omics

Highlights

- de Bruijn graph-based sequence assembly approaches have been an essential part of the broad application of sequencing methods, especially in microbiome research.
- de Bruijn graphs can be used to efficiently represent sequencing data in a format that is highly scalable and can be extended and modified to address different research questions.

283

Edge minimization in de Bruijn graphs

Uwe Baier*, Thomas Büchler*, Enno Ohlebusch*, Pascal Weber*
*Institute of Theoretical Computer Science
Ulm University, D-89069 Ulm, Germany
{uwe.baier,thomas.buechler,enno.ohlebusch,pascal-1.weber}@uni-ulm.de

Abstract

This paper introduces the de Bruijn graph edge minimization problem, which is related to the compression of de Bruijn graphs: find the order-k de Bruijn graph with minimum edge count among all orders. We describe an efficient algorithm that solves this problem. Since the edge minimization problem is connected to the BWT compression technique called "tunneling", the paper also describes a way to minimize the length of a tunneled BWT in such a way that useful properties for sequence analysis are preserved. Although being a restriction, this is significant progress towards a solution to the open problem of finding optimal disjoint blocks that minimize space, as stated in Alanko et al. (DCC 2019).

Introduction

284

Space-Efficient Merging of Succinct de Bruijn Graphs

Lavinia Egidi[1][0000−0002−9745−0942], Felipe A. Louza[2][0000−0003−2931−1470], and Giovanni Manzini[1,3][0000−0002−5047−0196]

[1] University of Eastern Piedmont, Alessandria, Italy
{lavinia.egidi, giovanni.manzini}@uniupo.it
[2] Department of Computing and Mathematics, University of São Paulo, Brazil
louza@usp.br
[3] IIT CNR, Pisa Italy

Abstract. We propose a new algorithm for merging succinct representations of *de Bruijn* graphs introduced in [Bowe *et al.* WABI 2012]. Our algorithm is based on the lightweight BWT merging approach by Holt and McMillan [Bionformatics 2014, ACM-BCB 2014]. Our algorithm has the same asymptotic cost of the state of the art tool for the same problem presented by Muggli *et al.* [bioRxiv 2017, Bioinformatics 2019], but it uses less than half of its working space. A novel important feature of our algorithm, not found in any of the existing tools, is that it can compute the *Variable Order* succinct representation of the union graph within the same asymptotic time/space bounds.

285

CALCOLO DEGLI OVERLAP

286

LCP

LCP (Longest Common Prefix) Array è stato introdotto da Myers e Manber nel 1993 come struttura ausiliaria per il Suffix Array.

DEF: dato un testo T ($-terminato) di n caratteri, LCP Array *LCP* è un array di n interi da 1 a n, tale che *LCP*[i] = ℓ se e solo se ℓ è la lunghezza del più lungo prefisso comune tra T[S[i]:] e T[S[i-1]:].
LCP[1] = -1 *by default*.

287

LCP

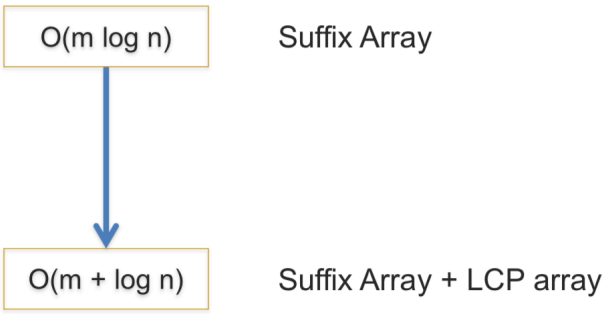
LCP (Longest Common Prefix) Array è stato introdotto da Myers e Manber nel 1993 come struttura ausiliaria per il Suffix Array.

Kasai et al. (2001)
LCP array da Suffix Array in tempo lineare

DEF
Arra
LCP
prefisso comune tra T[S[i]:] e T[S[i-1]:].
LCP[1] = -1 *by default*.

288

Pattern matching



un testo T lungo n, un pattern P lungo m

289

Indicizzazione per una collezione di stringhe

- ✓ Suffix Tree, Suffix Array, BWT e LCP Array generalizzati a una collezione di testi

290

Louza *et al. Algorithms Mol Biol* (2020) 15:18
https://doi.org/10.1186/s13015-020-00177-y

Algorithms for Molecular Biology

SOFTWARE ARTICLE Open Access

gsufsort: constructing suffix arrays, LCP arrays and BWTs for string collections

Felipe A. Louza[1*], Guilherme P. Telles[2], Simon Gog[3], Nicola Prezza[4] and Giovanna Rosone[5*]

Abstract

Background: The construction of a suffix array for a collection of strings is a fundamental task in Bioinformatics and in many other applications that process strings. Related data structures, as the Longest Common Prefix array, the Burrows–Wheeler transform, and the document array, are often needed to accompany the suffix array to efficiently solve a wide variety of problems. While several algorithms have been proposed to construct the suffix array for a single string, less emphasis has been put on algorithms to construct suffix arrays for string collections.

Result: In this paper we introduce `gsufsort`, an open source software for constructing the suffix array and related data indexing structures for a string collection with N symbols in $O(N)$ time. Our tool is written in `ANSI/C` and is based on the algorithm gSACA-K (Louza et al. in Theor Comput Sci 678:22–39, 2017), the fastest algorithm to construct suffix arrays for string collections. The tool supports large fasta, fastq and text files with multiple strings as input. Experiments have shown very good performance on different types of strings.

Conclusions: `gsufsort` is a fast, portable, and lightweight tool for constructing the suffix array and additional data structures for string collections.

Keywords: Suffix array, LCP array, Burrows–Wheeler transform, Document array, String collections

291

Indicizzazione per una collezione di stringhe

- ✓ Suffix Tree, Suffix Array, BWT e LCP Array generalizzati a una collezione di testi
- ✓ Riconoscimento di overlaps all'interno di una collezione di testi (*reads*)

SA+LCP+BWT→ overlap in tempo lineare

292

Anche la BWT consente di gestire i k-mer (dBG)

Space-efficient construction of succinct de Bruijn graphs

Felipe A. Louza

University of São Paulo, Brazil

Joint work with
Lavinia Egidi and Giovanni Manzini.

293

C. Hohlweg, C. Reutenauer
Lyndon words, permutations and trees
Theor. Comput. Sci., 307 (1) (2003), pp. 173-178

FATTORIZZAZIONI DI LYNDON E SA

294

FATTORIZZAZIONI DI LYNDON E BWT

295

FATTORIZZAZIONI DI LYNDON E OVERLAP

296

KFinger: Capturing Overlaps between Long Reads by Using Lyndon Fingerprints*

Paola Bonizzoni[1][0000−0001−7289−4988], Alessia Petescia[1][0000−0002−9000−3664], Yuri Pirola[1][0000−0002−8479−7592], Raffaella Rizzi[1][0000−0001−9730−7516], Rocco Zaccagnino[2][0000−0002−9089−5957], and Rosalba Zizza[2][0000−0001−9144−3074]

[1] Dip. di Informatica, Sistemistica e Comunicazione, University of Milano-Bicocca, viale Sarca 336, 20126 Milan, Italy
paola.bonizzoni@unimib.it, a.petescia@campus.unimib.it, yuri.pirola@unimib.it, raffaella.rizzi@unimib.it
[2] Dip. di Informatica, University of Salerno, via Giovanni Paolo II 132, 84084 Fisciano, Italy
{rzaccagnino, rzizza}@unisa.it

Abstract. Detecting common regions and overlaps between DNA sequences is crucial in many Bioinformatics tasks. One of them is genome assembly based on the use of the overlap graph which is constructed by detecting the overlap between genomic reads. When dealing with long reads this task is further complicated by the length of the reads and the high sequencing error rate. This paper proposes a novel alignment-free method for detecting the overlaps in a set of long reads which exploits a signature (called *fingerprint*) of reads built from a factorization of the read based on the notion of Lyndon words. The method has been implemented in the tool `KFinger` and tested over a simulated and a real PacBio HiFi dataset of genomic reads; its results have been compared with the well-known aligner `Minimap2`. `KFinger` is available at `https://github.com/AlgoLab/kfinger`.

297

Can We Replace Reads by Numeric Signatures? Lyndon Fingerprints as Representations of Sequencing Reads for Machine Learning

Paola Bonizzoni[1](✉), Clelia De Felice[3], Alessia Petescia[1], Yuri Pirola[1], Raffaella Rizzi[1], Jens Stoye[2], Rocco Zaccagnino[3], and Rosalba Zizza[3]

Abstract. Representations of biological sequences facilitating sequence comparison are crucial in several bioinformatics tasks. Recently, the Lyndon factorization has been proved to preserve common factors in overlapping reads [6], thus leading to the idea of using factorizations of sequences to define measures of similarity between reads. In this paper we propose as a signature of sequencing reads the notion of *fingerprint*, i.e., the sequence of lengths of consecutive factors in Lyndon-based factorizations of the reads. Surprisingly, fingerprints of reads are effective in preserving sequence similarities while providing a compact representation of the read, and so, k-mers extracted from a fingerprint, called k-fingers, can be used to capture sequence similarity between reads.

We first provide a probabilistic framework to estimate the behaviour of fingerprints. Then we experimentally evaluate the effectiveness of this representation for machine learning algorithms for classifying biological sequences. In particular, we considered the problem of assigning RNA-Seq reads to the most likely gene from which they were generated. Our results show that fingerprints can provide an effective machine learning interpretable representation, successfully preserving sequence similarity.

298

- The read-gene classification problem asks for the classification of each read of a given set of RNA-Seq reads to its putative origin gene. Instead, the chimeric reads detection problem can be seen as a generalization of read-gene classification since it requires to assign a chimeric read to two fused genes (instead of a single gene).

Numeric Lyndon-based feature embedding of sequencing reads for machine learning approaches

P. Bonizzoni [a], M. Costantini [a], C. De Felice [c], A. Petescia [a], Y. Pirola [a], M. Previtali [a], R. Rizzi [a], J. Stoye [b], R. Zaccagnino [c], R. Zizza [c]

Show more

https://doi.org/10.1016/j.ins.2022.06.005

Abstract

Feature embedding methods have been proposed in the literature to represent sequences as numeric vectors to be used in some bioinformatics investigations, such as family classification and protein structure prediction.

Recent theoretical results showed that the well-known Lyndon factorization preserves common factors in overlapping strings [1]. Surprisingly, the *fingerprint* of a sequencing read, which is the sequence of lengths of consecutive factors in variants of the Lyndon factorization of the read, is effective in capturing sequence similarities, suggesting it as basis for the definition of novel representations of sequencing reads.

We propose a novel feature embedding method for Next-Generation Sequencing (NGS) data using the notion of *fingerprint*. We provide a theoretical and experimental framework to estimate the behaviour of fingerprints and of the k-mers extracted from it, called *k-fingers*, as possible feature embeddings for sequencing reads. As a case study to assess the effectiveness of such embeddings, we use fingerprints to represent RNA-Seq reads in order to assign them to the most likely gene from which they originated as fragments of transcripts of the gene.

We provide an implementation of the proposed method in the tool *lyn2vec*, which produces Lyndon-based feature embeddings of sequencing reads.

299

Appunti di viaggio (e di progetto)

300

Zielezinski *et al. Genome Biology* (2017) 18:186
DOI 10.1186/s13059-017-1319-7

Genome Biology

REVIEW Open Access

Alignment-free sequence comparison: benefits, applications, and tools

Andrzej Zielezinski[1], Susana Vinga[2], Jonas Almeida[3] and Wojciech M. Karlowski[1*]

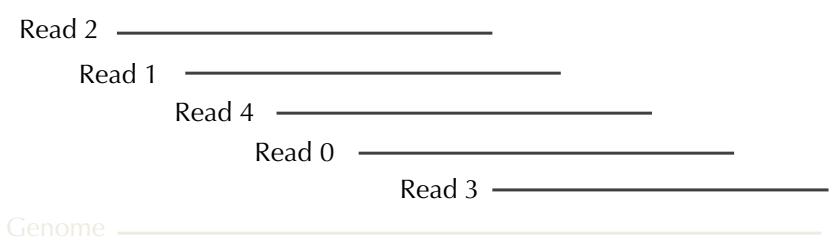
Abstract

Alignment-free sequence analyses have been applied to problems ranging from whole-genome phylogeny to the classification of protein families, identification of horizontally transferred genes, and detection of recombined sequences. The strength of these methods makes them particularly useful for next-generation sequencing data processing and analysis. However, many researchers are unclear about how these methods work, how they compare to alignment-based methods, and what their potential is for use for their research. We address these questions and provide a guide to the currently available alignment-free sequence analysis tools.

301

Read mapping again

We need a *heuristic* for multiple alignment of overlapping reads. (Harder than it seems!)



302

Question

If we have 30,000 reads (e.g., a bacterial dataset), then we are going to have to perform $(30{,}000)^2$ = 900 *million* alignments. This is before we have to work with the overlap network!

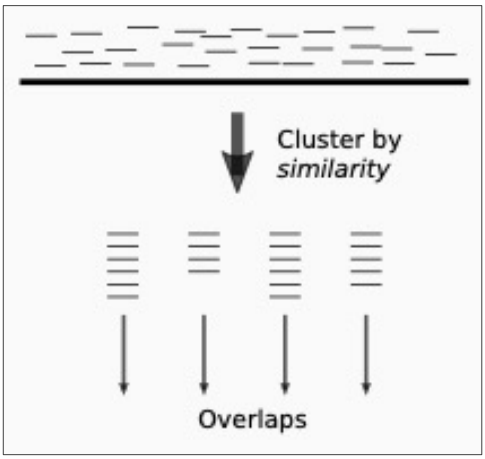
Key Point: we can hit the boundary of what a (fast, modern) computer can do very quickly …

303

Finding similar strings before overlapping

Idea: make a quick decision at the start to find strings that may be similar.

STOP: any ideas on how we could we quickly cluster reads based on similarity?



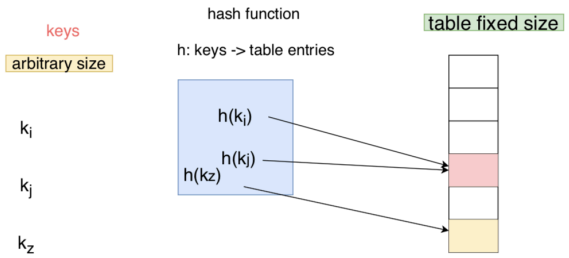
304

The Brilliant Insight

The more two strings *overlap*, the more *k*-mers that they will share.

305

Hashing: accesso costante al dato



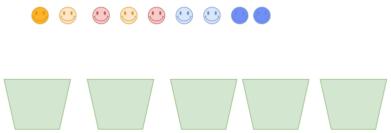
se oggetti simili collidono nella stessa posizione… allora ho un metodo veloce (costante!) per stabilire somiglianza tra oggetti (opportuna collisione diventa strumento positivo)

306

Hashing e alcune misure di somiglianza

- Idea di processare buckets, contenitori dove elementi finiscono con stesso hash
- misure di somiglianza: distanza di Jaccard, distanza di Hamming Distance e distanza di edit

Trilioni di buckets e dobbiamo hashare elementi nei buckets



307

Nozioni: Jaccard distance

La distanza d or differenza tra due stringhe x, y, mentre la somiglianza è data da $1 - d$.

misura di similarità tra insiemi (la applico ai due insiemi di kmer di due read per dedurre l'overlap)

Esempi:

- Jaccard distance tra due insiemi A e B definita come: $J(A, B) = \frac{A \cap B}{A \cup B}$
- Hamming distance per due stringhe di lunghezza n, numero di posizioni in cui differiscono

308

Hashing e alcune misure di somiglianza

- Idea di processare buckets, contenitori dove elementi finiscono con stesso hash
- misure di somiglianza: distanza di Jaccard, distanza di Hamming Distance e distanza di edit

Oggetti che hanno stesso hash hanno alta probabilità di essere simili

non confronto tra loro

309

(localmente sensibile)

Una famiglia di funzioni hash è **locality sensitive** se elementi simili hanno alta probabilità di avere stessa funziona di Hash

Bioinformatics, 35, 2019, i127–i135
doi: 10.1093/bioinformatics/btz354
ISMB/ECCB 2019

Locality-sensitive hashing for the edit distance

Guillaume Marçais*, Dan DeBlasio, Prashant Pandey and Carl Kingsford*

Computational Biology Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA

*To whom correspondence should be addressed.

Abstract

Motivation: Sequence alignment is a central operation in bioinformatics pipeline and, despite many improvements, remains a computationally challenging problem. Locality-sensitive hashing (LSH) is one method used to estimate the likelihood of two sequences to have a proper alignment. Using an LSH, it is possible to separate, with high probability and relatively low computation, the pairs of sequences that do not have high-quality alignment from those that may. Therefore, an LSH reduces the overall computational requirement while not introducing many false negatives (i.e. omitting to report a valid alignment). However, current LSH methods treat sequences as a bag of *k*-mers and do not take into account the relative ordering of *k*-mers in sequences. In addition, due to the lack of a practical LSH method for edit distance, in practice, LSH methods for Jaccard similarity or Hamming similarity are used as a proxy.
Results: We present an LSH method, called Order Min Hash (OMH), for the edit distance. This method is a refinement of the minHash LSH used to approximate the Jaccard similarity, in that OMH is sensitive not only to the *k*-mer contents of the sequences but also to the relative order of the *k*-mers in the sequences. We present theoretical guarantees of the OMH as a gapped LSH.
Availability and implementation: The code to generate the results is available at http://github.com/

310

Local sensitive Hashing

(localmente sensibile)

Una famiglia di funzioni hash è **locality sensitive** se elementi simili hanno alta probabilità di avere stessa funziona di Hash

Definizione

LSH per una relazione s() di somiglianza è una famiglia F di hash functions tale che:

$Pr_{h \in F}[h(x) = h(y)] = s(x, y)$, per ogni x, y nel dominio di F.

Come stimiamo $s(x, y)$?

311

Local sensitive Hashing:come stimiamo $s(x, y)$?

kmer comuni
—> parte comune

Confronto tra x, y usando lo sketch di x, y:

sketch$(x) =< h_1(x) \cdots h_t(x) >$,

sketch$(y) =< h_1(y) \cdots h_t(y) >$, usando t hash functions applicate a x, y, $h_1() \cdots h_t()$.

stima di $s(x, y)$

$s(x, y) = \frac{\#h_i(x)=h_i(y)}{t}$, per ogni x, y nel dominio di F dove la collisione indica somiglianza

costruzione di F?

non confronto i k-mer a coppie.
Uso sketch e indicizzazione dei k-mer con tabelle hash

Obiettivo: definire F in modo tale che il numero di volte due oggetti condividono le collisioni è la somiglianza...

Se le t hash hanno lo stesso valore per x e y, allora x e y sono simili

312

Le idee principali di LSH

La procedura LSH è un metodo di riduzione delle dimensioni
Step 1: le sequenze (o parti di esse) vengono riassunte in sketches - cosa è uno sketch? E' una rappresentazione più piccola della sequenza originaria
Step 2: confrontabdo gli sketches come keys di tabelle hash, il software trova coppie di sequenze che sono simili con alta probabilità
Note: minHash LSH (Broder, 1997) per la Jaccard distance o per Hamming distance come approssimazione della somiglianza.

313

MinHash sketch

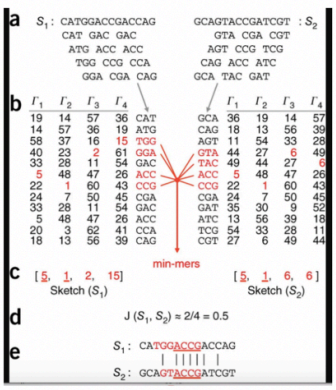
definition: LSH for a similarity is a family of hash functions that satisfies the following:
$Pr(h(x) = h(y)) = s(x, y)$
- minHash LSH for the Jaccard distance between two sets A and B is a family of hash functions such that
$Pr[h(A) = h(B)] = J(A, B)$, for A and B sets.

314

MinHash sketch: example of applications

creation of a MinHash sketch for a DNA sequence S

- given $k = 3$, S_1 and S_2 have 12 k-mers
- k-mers are converted into integer fingerprints
- 4 hash functions Γ_1, Γ_2, Γ_3, Γ_4, and for each we choose the minimum, producing the sketch
- the sketch is smaller in size than the collection of k-mers
- The fraction of entries shared between the sketches of two sequences S1 and S2 (0.5) serves as an estimate of their true Jaccard similarity (0.22)
- the size of the sketch is used to obtain accurate estimates



questi sketch rappresentano i k-mer
Quindi usiamo minhash per indicizzare i k-mer. Altro metodo sono i Bloom Filter

315

An example of application

Ondov *et al. Genome Biology* (2016) 17:132
DOI 10.1186/s13059-016-0997-x

Genome Biology

SOFTWARE Open Access

Mash: fast genome and metagenome distance estimation using MinHash

Brian D. Ondov[1], Todd J. Treangen[1], Páll Melsted[2], Adam B. Mallonee[1], Nicholas H. Bergman[1], Sergey Koren[3] and Adam M. Phillippy[3*]

Abstract

Mash extends the MinHash dimensionality-reduction technique to include a pairwise mutation distance and *P* value significance test, enabling the efficient clustering and search of massive sequence collections. Mash reduces large sequences and sequence sets to small, representative sketches, from which global mutation distances can be rapidly estimated. We demonstrate several use cases, including the clustering of all 54,118 NCBI RefSeq genomes in 33 CPU h; real-time database search using assembled or unassembled Illumina, Pacific Biosciences, and Oxford Nanopore data; and the scalable clustering of hundreds of metagenomic samples by composition. Mash is freely released under a BSD license (https://github.com/marbl/mash).

Keywords: Comparative genomics, Genomic distance, Alignment, Sequencing, Nanopore, Metagenomics

316

Mash: the idea

- Take genomes that need to be pairwise compared in a fast way.
- For each genome extracts the k-mers and hash them via single hash function (canonical k-mers: the smallest lexicographic one by the forward and the reverse complement)
- Fix a sketch size s: output the s smallest hashes output over all k-mers
- Compression: 618 Gbp of genomic sequence and sketches total only 93 MB

317

k-mer counting

(conteggio, rappresentazione con Bloom Filter, Hash, minhash, BWT)

318

minimizer
smallest ciclic permutation of a kmer (also w.r.t. reverse and complement)

CGTTGATCAATTTG

K=4;

AATT è il più piccolo?

Ma il suo reverse-complement è AATT.

Scelgo ATTT: ha come reverse-complement è AAAT.
smallest ciclic permutation of a kmer (also w.r.t. reverse and complement)

319

minimizer
smallest ciclic permutation of a kmer (also w.r.t. reverse and complement)

CGTTGATCAATTTG

Se 2 read sono in overlap, se il minimizer cade all'interno, decido overlap!

K=4;

AATT è il più piccolo?

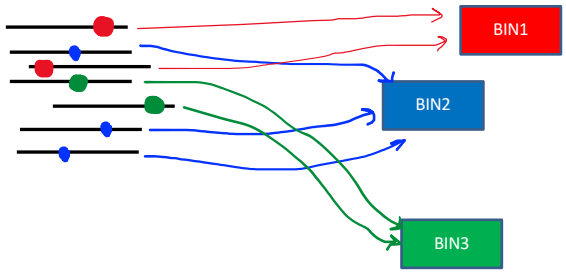
Ma il suo reverse-complement è AATT.

Scelgo ATTT: ha come reverse-complement è AAAT.
smallest ciclic permutation of a kmer (also w.r.t. reverse and complement)

320

Uso (in KMC) – kmer counting

Se ho varie read, ho i rispettivi minimizer (etichette di BIN1, BIN2, BIN3), uso i minimizer per tenere traccia dei kmer (andranno nello stesso contenitore)



dato k, posso predeterminare i BIN (considero i kmer possibili in ordine lessicografico) – ottimizzato per farlo on-line…

e poi posso fare il calcolo in parallelo sui BIN

bilanciamento dei BIN; scelta del k; non va bene per tutti i contesti

321

- Experimental evaluation

Table 4. *k*-mers counting results for *H.sapiens* 2

Algorithm	$k=28$			$k=55$		
	RAM	Disk	Time	RAM	Disk	Time
SSD						
Jellyfish 2	62	0	3212	*Out of memory*		
KAnalyze	*Out of disk (>650 GB)*			*Unsupported k*		
DSK	6	263	5487	6	256	7732
Turtle	*Out of memory*			*Out of memory*		
MSPKC	*Out of time (>10 h)*			*Out of time (>10 h)*		
KMC 1	17	396	2998	*Out of disk (>650 GB)*		
KMC 2 (12 GB)	12	101	1615	13	70	2038
KMC 2 (6 GB)	6	101	1706	13	70	2446
KMC 2 (4 GB)	4	101	1843	4	70	2802
HDD						
Jellyfish 2	62	0	3231	*Out of memory*		
DSK	6	263	18 493	6	256	22 432
KMC 1	17	396	4898	*Out of disk (>650 GB)*		
KMC 2 (12 GB)	12	101	2259	13	70	2640
KMC 2 (4 GB)	4	101	2707	4	70	3471

Timings in seconds and RAM/disk consumption in GB.

Table 3. *k*-mers counting results for *M.balbisiana*

Algorithm	$k=28$			$k=55$		
	RAM	Disk	Time	RAM	Disk	Time
SSD						
Jellyfish 2	17	0	1080	26	0	853
Jellyfish 2-BF	*Out of memory*			56	30	2865
KAnalyze	9	354	8249	—	—	—
DSK	6	164	2356	6	138	2962
Turtle	46	0	1484	*Out of memory*		
MSPKC	10	185	8729	*Out of time (>10 h)*		
KMC 1	13	165	1229	15	279	2622
KMC 2 (12 GB)	12	41	755	12	29	834
KMC 2 (6 GB)	6	41	685	6	29	895
KMC 2 (4 GB)	4	41	833	4	29	896
KMC 2 (in-mem)	49	0	663	38	0	780
HDD						
Jellyfish 2	17	0	1115	26	0	881
DSK	6	164	6216	6	138	7228
Turtle	46	0	1498	*Out of memory*		
MSPKC	10	185	12 152	*Out of time (>10 h)*		
KMC 1	13	165	2194	15	279	3367
KMC 2 (12 GB)	12	41	960	12	29	1041
KMC 2 (4 GB)	4	41	1051	4	29	1207

Timings in seconds and RAM/disk consumption in GB.

322

Using Minimizers for Read Mapping?

We can pre-process the genome by building an index containing the *k*-mer minimizer of each window for some window length *w*.

Genome

323

Using Minimizers for Read Mapping?

We can pre-process the genome by building an index containing the *k*-mer minimizer of each window for some window length *w*.

Then, for each read, we compute its minimizers.

Genome

324

Using Minimizers for Read Mapping?

We can pre-process the genome by building an index containing the *k*-mer minimizer of each window for some window length *w*.

Then, for each read, we compute its minimizers.

Genome

STOP: How will we know if a (long) read aligns well to the genome?

325

Using Minimizers for Read Mapping?

We can pre-process the genome by building an index containing the *k*-mer minimizer of each window for some window length *w*.

Then, for each read, we compute its minimizers.

Genome

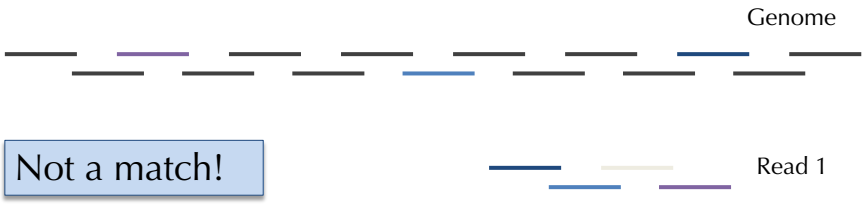
Answer: If it matches multiple minimizers that come from *a small region* of the genome.

326

Using Minimizers for Read Mapping?

We can pre-process the genome by building an index containing the *k*-mer minimizer of each window for some window length *w*.

Then, for each read, we compute its minimizers.



Not a match!

327

Using Minimizers for Read Mapping?

We can pre-process the genome by building an index containing the *k*-mer minimizer of each window for some window length *w*.

Then, for each read, we compute its minimizers.

Possible match – align the read!

328

Using Minimizers for Read Mapping?

This is the approach applied by "Minimap".

www.ncbi.nlm.nih.gov › pmc › articles › PMC4937194

Minimap and miniasm: fast mapping and de novo ... - NCBI - NIH

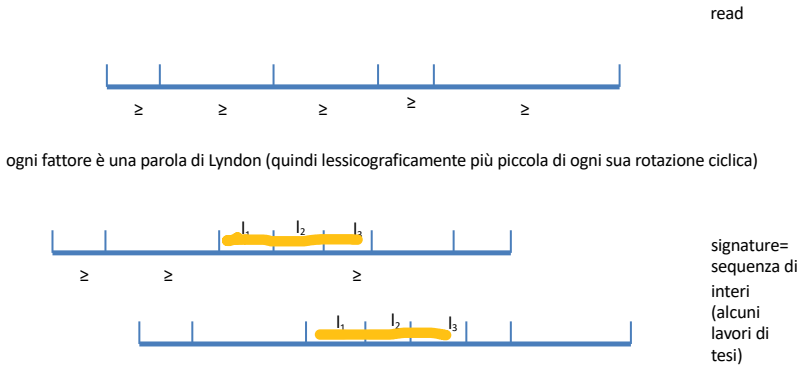
Mar 19, 2016 — Results: We present a new **mapper**, **minimap** and a de novo assembler, miniasm, ... Notably, our tool **minimap is** a raw **read** overlapper and miniasm **is** an assembler. We **do** ... **Minimap is** heavily influenced by all these **works**.

by H Li · 2016 · Cited by 462 · Related articles

Possible match – align the read!

329

Signature con Fattorizzazione Lyndon



ogni fattore è una parola di Lyndon (quindi lessicograficamente più piccola di ogni sua rotazione ciclica)

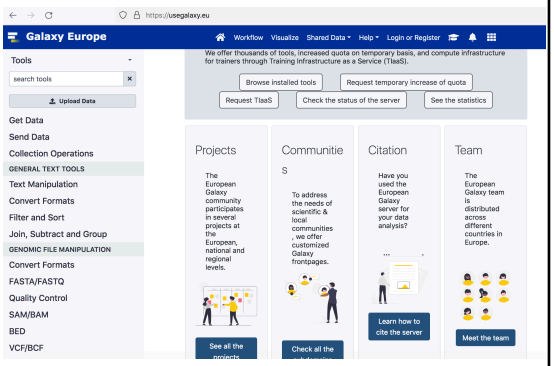
signature= sequenza di interi (alcuni lavori di tesi)

330

Per l'esame

Nei prossimi giorni

- elenco dei progetti con breve descrizione sarà aggiornato
- assignments e training
- scegliete quello che volete sviluppare
- anche in team
- non sparite ☺



331

Per l'esame

Sito

- elenco dei progetti con breve descrizione sarà aggiornato
- assignments e training
- scegliete quello che volete sviluppare
- anche in team
- non sparite ☺

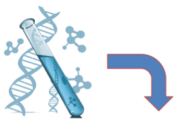
Date d'esame pubblicate:

- ci metteremo d'accordo per fare le presentazioni alla classe, orientativamente in quelle date
- (prenotatevi)

- scegliete qualcosa che vi incuriosisce …

332

Viaggio al centro del



+ algoritmi

Sequenziamento DNA

Terapia medica

è stato un bel viaggio…

333