CM122/222 Bioinformatics Algorithms

Discussion 1A

Shuwen Qiu

(Slides Prepared by Xuheng Li)

Outline

- Task: Multiple pattern matching problem
- Approaches
    - Brute-force pattern matching
    - Trie maching
    - Suffix trie, suffix tree and suffix array
    - Burrows-Wheeler Transform (BWT)

Task: Multiple pattern matching problem

- Find all occurrences of a collection of patterns in a text.
- Input:
  - A string *Text;*
  - A collection *Patterns* containing shorter strings.
- Output: All starting positions in *Text* where a string from *Patterns* appears as a substring.
- Example:
  - Input:
    - Text = AATCGGGTTCAATCGGGGT
    - Patterns = {ATCG, GGGT}
  - Output:
    - ATCG: 1, 11; GGGT: 4, 15.

Brute-force pattern matching

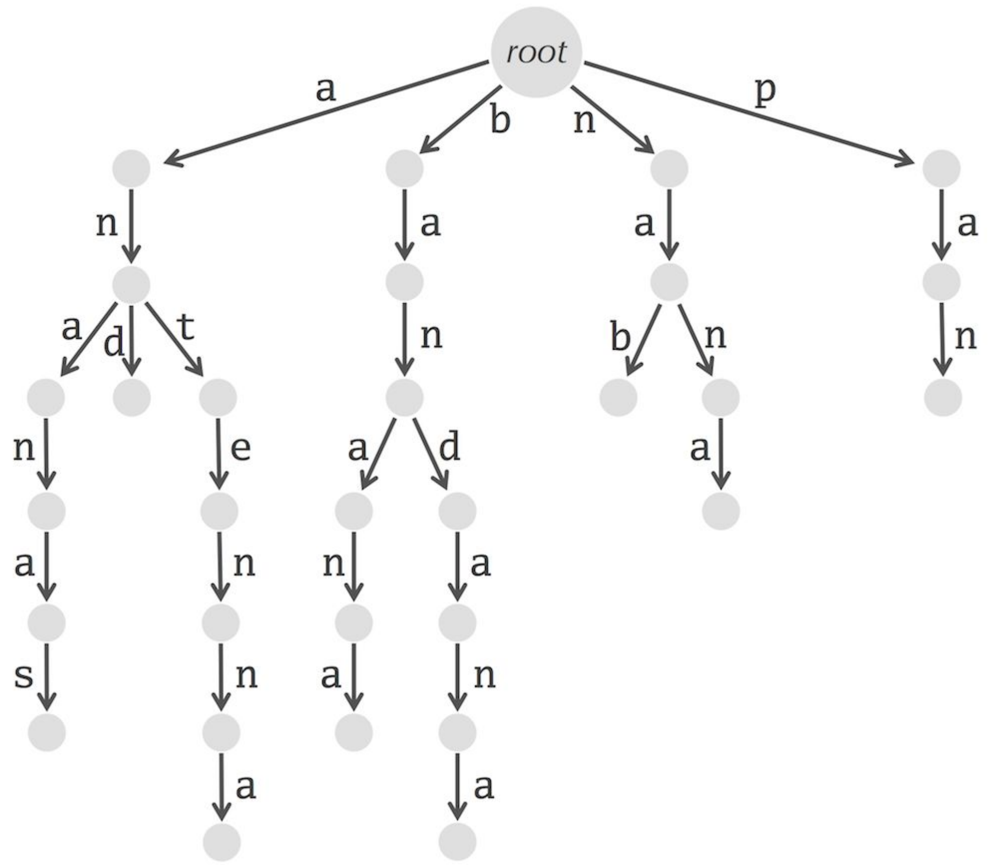
- Method:
  - Slide each *Pattern* along *Text*;
  - Check whether substring starting at each position matches *Pattern*.
- Runtime
  - Suppose there are n patterns;
  - Maximum pattern length is m;
  - Sum of lengths of all patterns is M;
  - Length of Text is L.
  - Time complexity: $O(ML)$.

Trie Matching

Trie

- Components of a **trie**:
    - A trie is a tree;
    - Nodes, including a **root** node and **leaf** nodes.
    - Edges, each labelled by a letter of the alphabet.
- Properties:
    - Different edges coming out of a node have different labels;
    - Each path from the root node to a leaf node represents a pattern.

Trie



Textbook Chapter 9.3

- Patterns encoded in the trie:
  - ananas
  - and
  - antenna
  - banana
  - bandana
  - nab
  - nana
  - pan

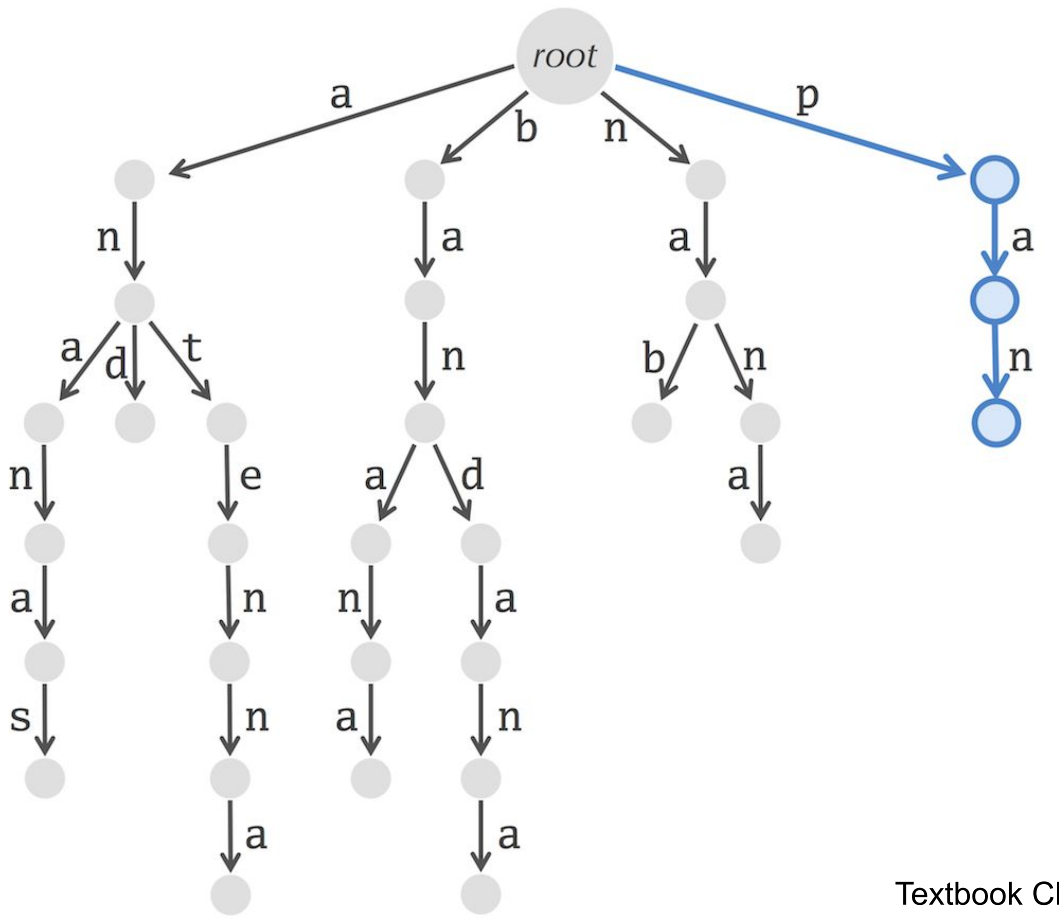
```
class Node
def TrieConstruct(Patterns):
    root = Node(0)
    for pattern in Patterns:
        current node <- root
        for each character in pattern:
            if character in current node.edges:
                current node <- root.edge2node[character]
            else:
                new node = Node(new id)
                current node.edges.append(charater)
                current node.edge2node[charater] = new node
                current node <- new node
    return root
```

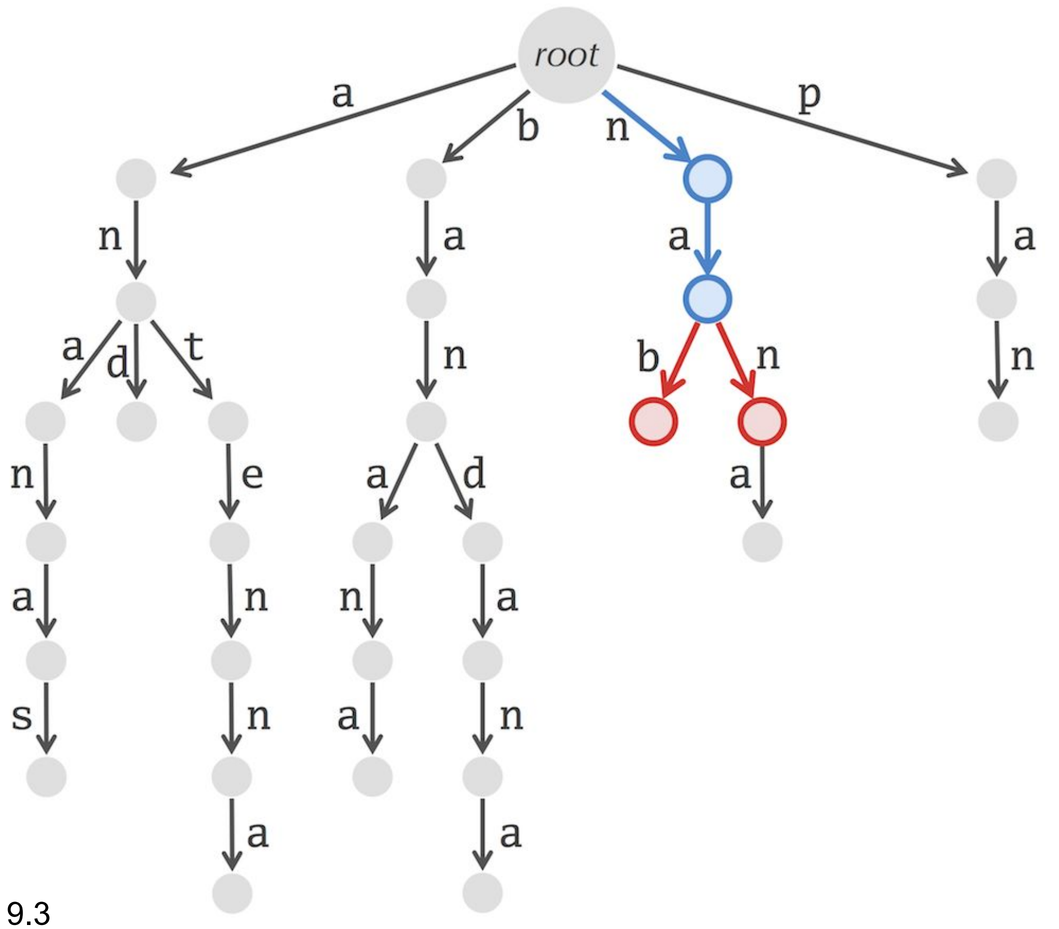
Trie construction

To add *Pattern* to the trie:

- Start with the root node.
- For each *letter* in *Pattern*:
    - If the current node has an outgoing edge labelled by *letter*, then proceed along this edge to another node;
    - Else, create a new node and an edge from the current node to the new node labelled by *letter*. Proceed to the new node.

Trie matching





Textbook Chapter 9.3

Text = panamabananas. Successful match from 1st letter of *Text*. No pattern matches from 3rd letter of *Text*.

Problem with Trie Matching

- What if one pattern is the prefix of another?
- Example: *Patterns* = {A, AA}
- What is the corresponding trie?
- Cannot determine the end of a pattern by whether a leaf node is reached!

Problem with Trie Matching

- What if one pattern is the prefix of another?
- Example: *Patterns* = {A, AA}
- What is the corresponding trie?
- Cannot determine the end of a pattern by whether a leaf node is reached!

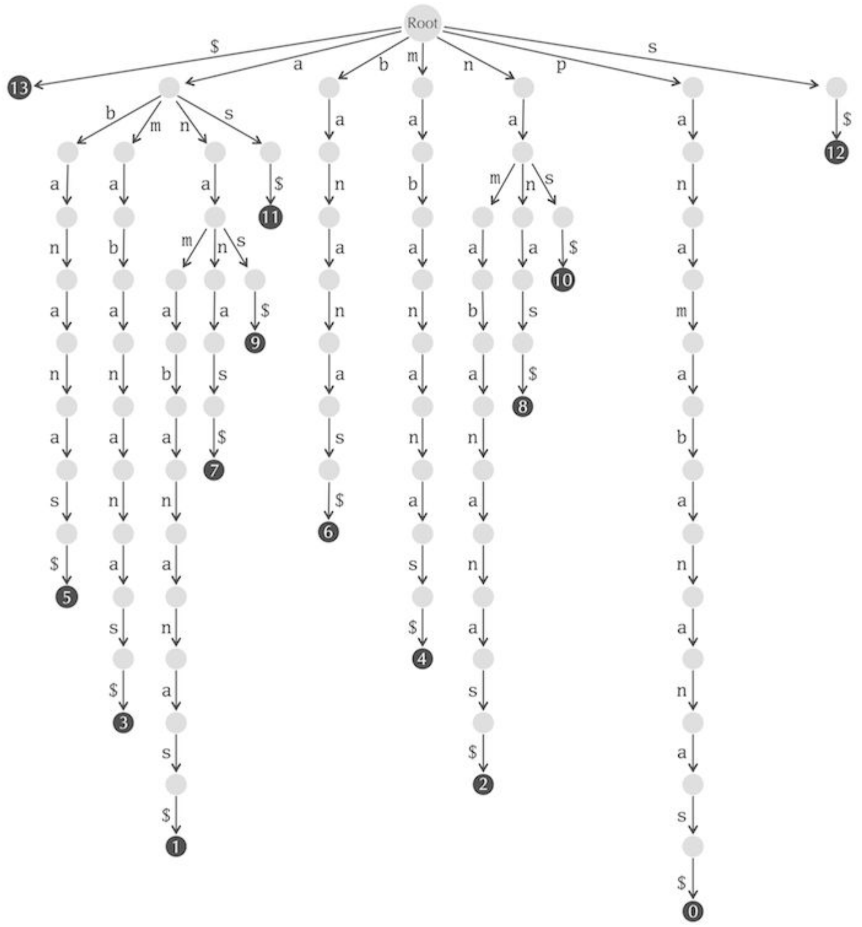
- Solution:
  - Append "$" to each pattern.
  - End of a pattern: node with an outgoing edge labelled "$".

Trie Matching

- Subroutine: find patterns that match with a prefix of *Text* (prefix matching).
  - Start with the root node.
  - For each *letter* of *Text*:
    - If the current node is a leaf node, then return the word spelled by the path from the root node to the current node.
    - If there is an outgoing edge from the current node labelled by *letter*, then proceed along the edge to another node.
    - Else, no pattern is found.
- Trie matching: Perform prefix matching for each suffix of *Text*.
- Runtime: $O(M + mL)$
- Memory: $O(M)$

Suffix Trie,
Suffix Tree,
Suffix Array

Suffix trie



Textbook Chapter 9.4

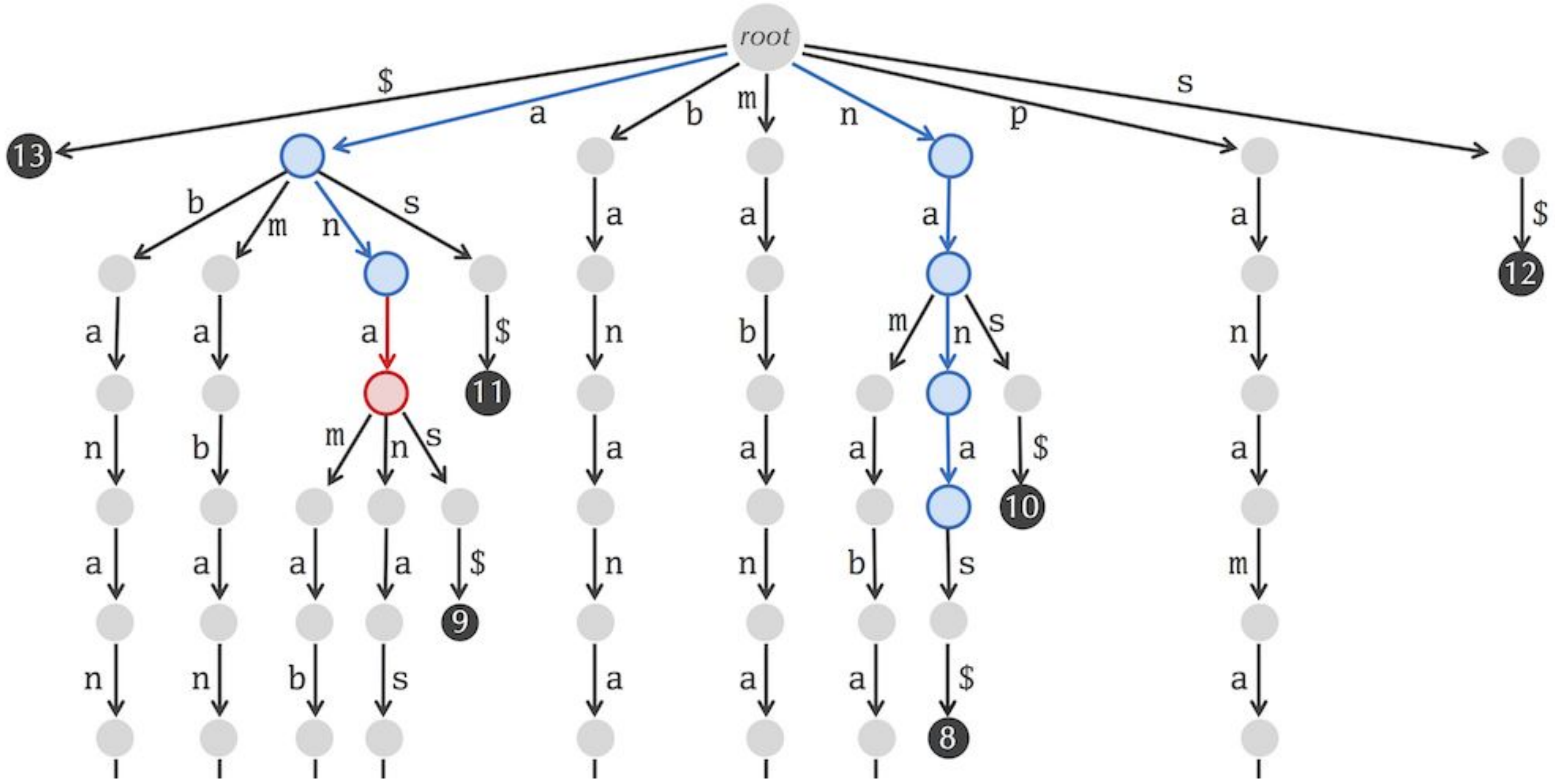
- A suffix trie is a trie that encodes all suffixes of Text (with “$” appended).
- Each leaf marks the starting position of the suffix.
- Why $: papa

```
class Node
    self.label
def SuffixTrieConstruct(Text):
    root = Node(0)
    for i from 0 to len(Text)-1:
        pattern = Text[i:]
        current node <- root
        for each character in pattern:
            if character in current node.edges:
                current node <- root.edge2node[character]
            else:
                new node = Node(new id)
                current node.edges.append(charater)
                current node.edge2node[charater] = new node
                current node <- new node
            if character == '$':
                current node.label = i
    return root
```

Suffix Trie Construct

```
for each suffix in Text:
    for each symbol in suffix:
        currentnode <- root node
        if symbol in node.edges:
            currentnode <- node.edge2node
        else:
            add new node
            currentnode <- new node
        if symbol == ‘$’:
            currentnode.label = suffix position
```

Suffix trie matching

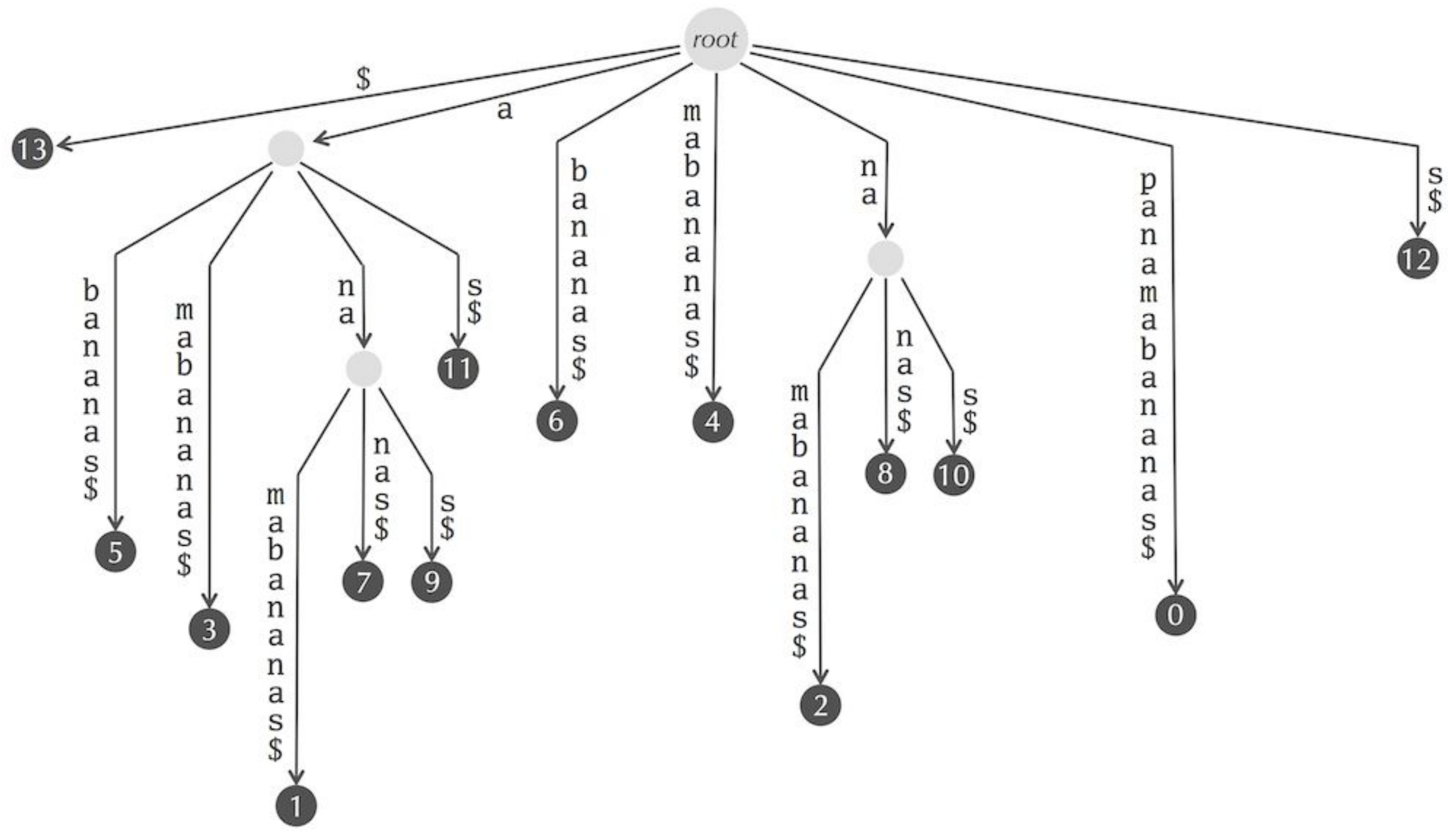


Textbook Chapter 9.4

Suffix trie matching

- Method:
  - Start from root node.
  - For each *letter* in *Pattern*:
    - If the current node has an outgoing edge labelled by *letter*, then proceed along the edge to another node;
    - Else, *Pattern* is not matched by any subsequence of *Text*.
  - Return **labels of all leaf nodes** under the current node.
- Runtime and memory when constructing suffix trie: $O(L^2)$.

Suffix tree



- Designed to save memory of suffix trie.
- From suffix **trie** to suffix **tree**: combine non-branching paths.
- For each edge, store only the **starting position** of the subsequence in *Text* and its **length**.
- Memory: $O(L)$.

Textbook Chapter 9.5

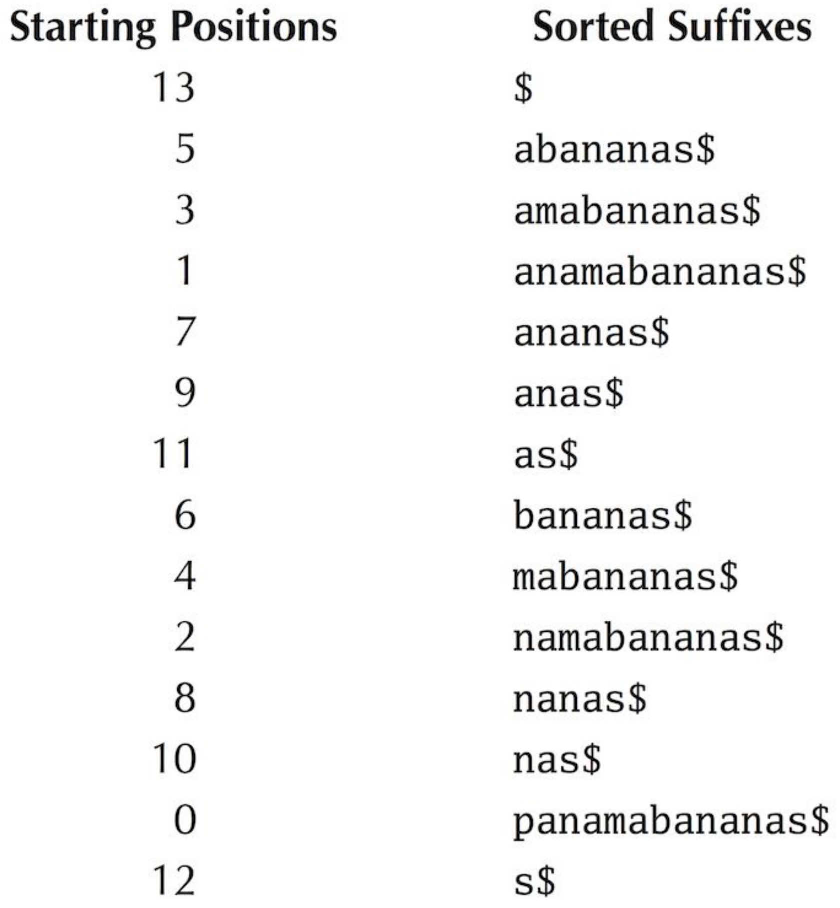
```
def TraverseTrie(root):
    node_list = [root]
    while len(node_list) > 0:
        current node = pop up first element of node list
        for edge in current node.edges:
            node_list.append(current node.edge2node[edge])
```

Suffix tree construction

Trie → Tree:

```
for each node in Trie:
    if number of edges > 1:
        for edge in edges:
            new_edge = []
            next_node = node.edge2node
            while number of edges of next node == 1:
                new_edge.append(edge)
                next_node = next_node.edge2node
            node.edge = new_edge
            node.edge2node = next_node
```

Suffix array



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

Textbook Chapter 9.6

Construction:

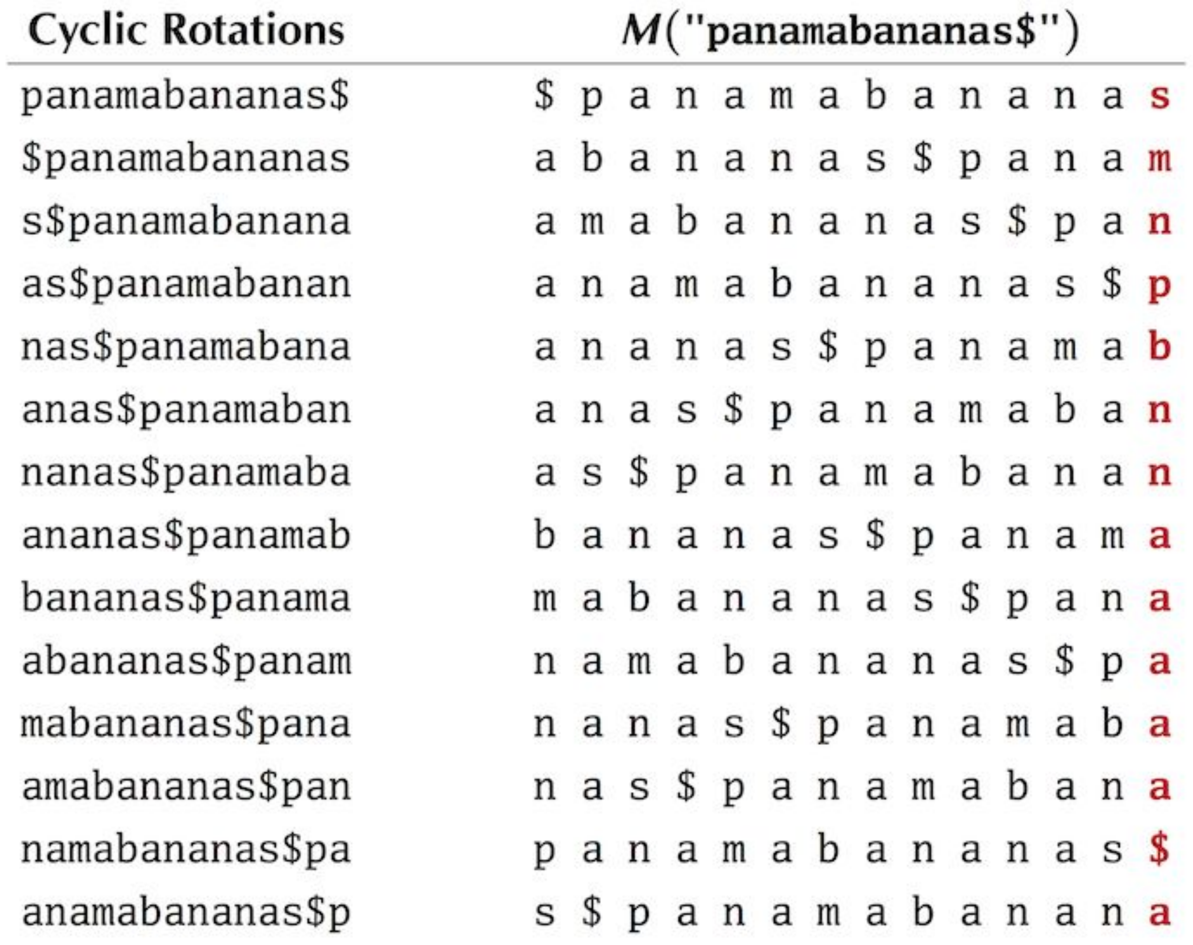
- Sort suffixes of *Text* lexicographically (“$” comes first);
- List starting positions.

Suffix array matching:

- Use **binary search** among all suffixes to find first occurrence of *Pattern* in suffix array;
- Use **binary search** after the first occurrence of *Pattern* to find its last occurrence of in suffix array.

Burrows-Wheeler Transform (BWT)

BWT construction



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

Textbook Chapter 9.7

- List all cyclic rotations;
- Sort cyclic rotations to get Burrows-Wheeler matrix (BWM).
- BWT = last column of BWM.

First-last property

$	p	a	n	a	m	a	b	a	n	a	n	a	s
a_1	b	a	n	a	n	a	s	$	p	a	n	a	m
a_2	m	a	b	a	n	a	n	a	s	$	p	a	n
a_3	n	a	m	a	b	a	n	a	n	a	s	$	p
a_4	n	a	n	a	s	$	p	a	n	a	m	a	b
a_5	n	a	s	$	p	a	n	a	m	a	b	a	n
a_6	s	$	p	a	n	a	m	a	b	a	n	a	n
b	a	n	a	n	a	s	$	p	a	n	a	m	a_1
m	a	b	a	n	a	n	a	s	$	p	a	n	a_2
n	a	m	a	b	a	n	a	n	a	s	$	p	a_3
n	a	n	a	s	$	p	a	n	a	m	a	b	a_4
n	a	s	$	p	a	n	a	m	a	b	a	n	a_5
p	a	n	a	m	a	b	a	n	a	n	a	s	$
s	$	p	a	n	a	m	a	b	a	n	a	n	a_6

Textbook Chapter 9.9

- **The k-th occurrence of a symbol in the first column and the k-th occurrence in the last column correspond to the same position in *Text*.**

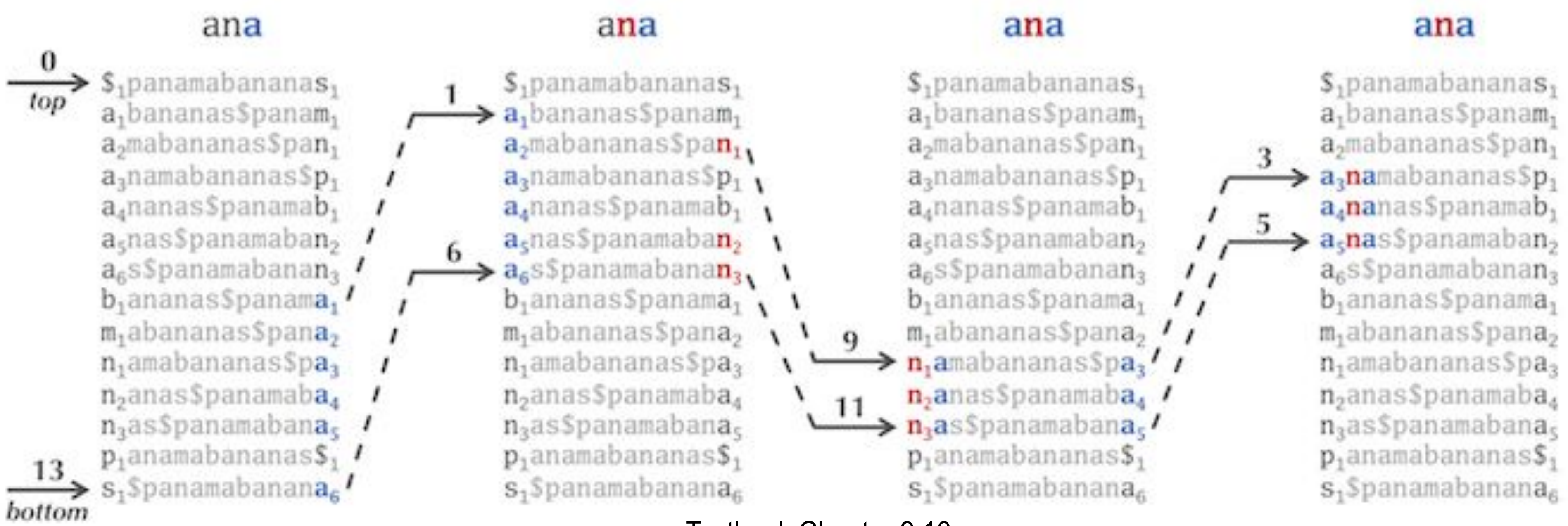
Why?

- Consider all occurrences of a symbol in the first column.
- After moving the symbol to the end of each of these rows, they will still be sorted.
- Moreover, they become the rows with the symbol in the last column.

Inverse BWT

- Example:
- enwvpeoseu$llt

BW matching



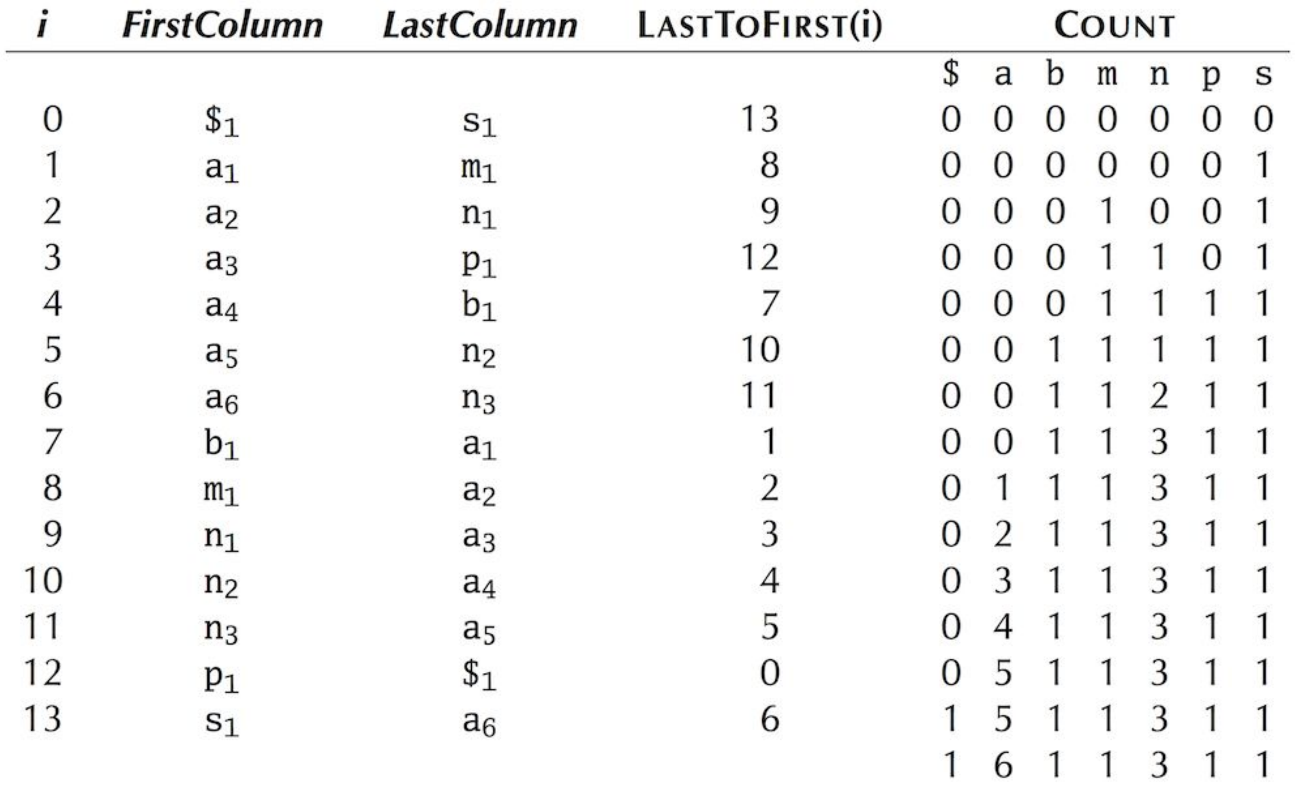
Textbook Chapter 9.10

- Observation: Consecutive occurrences of any symbol in the last column correspond to consecutive occurrences in the first column.
- Finding all occurrences reduced to finding first and last occurrences.
- Maintain two pointers *top* and *bottom*.

BW matching

```
top  = 0
bottom = len(Text)
for s in Pattern:
    if s in BWText[top:bottom]:
        topindex = first position
        bottomindex = last position
        top = last2first(topindex)
        bottom = last2first(bottomindex)
    else:
        return 0
return bottom-top+1
```

BW matching: speedup



i	*FirstColumn*	*LastColumn*	LASTTOFIRST(i)	COUNT						
				$	a	b	m	n	p	s
0	$\$_1$	s_1	13	0	0	0	0	0	0	0
1	a_1	m_1	8	0	0	0	0	0	0	1
2	a_2	n_1	9	0	0	0	1	0	0	1
3	a_3	p_1	12	0	0	0	1	1	0	1
4	a_4	b_1	7	0	0	0	1	1	1	1
5	a_5	n_2	10	0	0	1	1	1	1	1
6	a_6	n_3	11	0	0	1	1	2	1	1
7	b_1	a_1	1	0	0	1	1	3	1	1
8	m_1	a_2	2	0	1	1	1	3	1	1
9	n_1	a_3	3	0	2	1	1	3	1	1
10	n_2	a_4	4	0	3	1	1	3	1	1
11	n_3	a_5	5	0	4	1	1	3	1	1
12	p_1	$\$_1$	0	0	5	1	1	3	1	1
13	s_1	a_6	6	1	5	1	1	3	1	1
				1	6	1	1	3	1	1

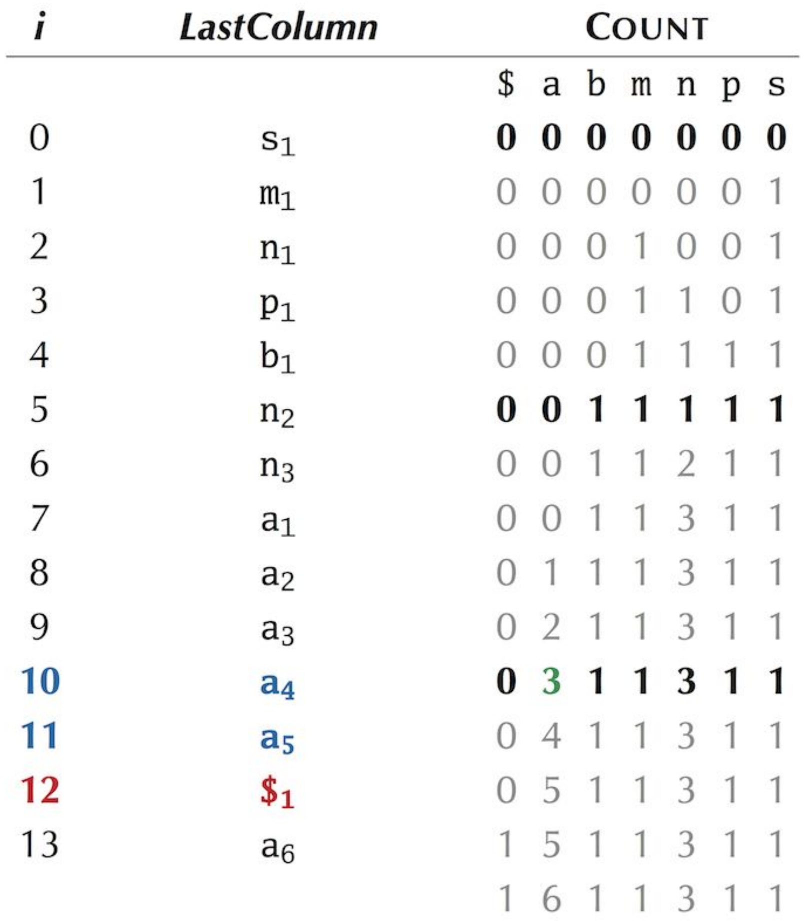
Textbook Chapter 9.11

- Count array:
- $Count_{symbol}(i, LastColumn)$: number of occurrences of *symbol* in the last column before the i-th row.

BW matching: speedup

- Consider update:
  - $top \leftarrow LastToFirst(\text{First occurrence of } symbol \text{ after } top \text{ in } LastColumn)$
- Righthand side is equal to:
  - First occurrence of $symbol$ in $FirstColumn + Count_{symbol}(top, LastColumn)$
- Use $FirstOccurrence(symbol)$ to represent the first occurrence of a symbol in the first column.
- Update of top:
  - $top \leftarrow FirstOccurrence(symbol) + Count_{symbol}(top, LastColumn)$
- Update of $bottom$:
  - $bottom \leftarrow FirstOccurrence(symbol) + Count_{symbol}(bottom + 1, LastColumn) - 1$

BW matching: further improvement



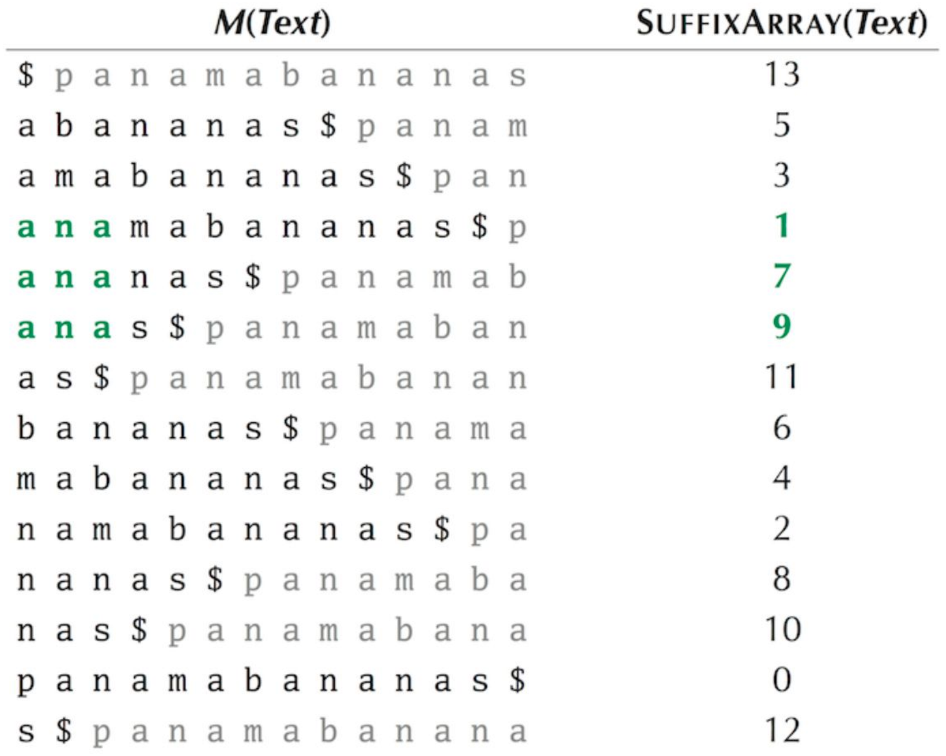
i	*LastColumn*	COUNT						
		$	a	b	m	n	p	s
0	s_1	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	m_1	0	0	0	0	0	0	1
2	n_1	0	0	0	1	0	0	1
3	p_1	0	0	0	1	1	0	1
4	b_1	0	0	0	1	1	1	1
5	n_2	**0**	**0**	**1**	**1**	**1**	**1**	**1**
6	n_3	0	0	1	1	2	1	1
7	a_1	0	0	1	1	3	1	1
8	a_2	0	1	1	1	3	1	1
9	a_3	0	2	1	1	3	1	1
10	a_4	**0**	**3**	**1**	**1**	**3**	**1**	**1**
11	a_5	0	4	1	1	3	1	1
12	$\$_1$	0	5	1	1	3	1	1
13	a_6	1	5	1	1	3	1	1
		1	6	1	1	3	1	1

Textbook Chapter 9.11

- Storing the entire count array is expensive!
- Solution:
  - Set up **checkpoints** at rows with indices being a multiple of C.
  - To obtain count array of any row, start with the closest checkpoint and count all occurrences of *symbol* until the desired row.

Where are the matched patterns?

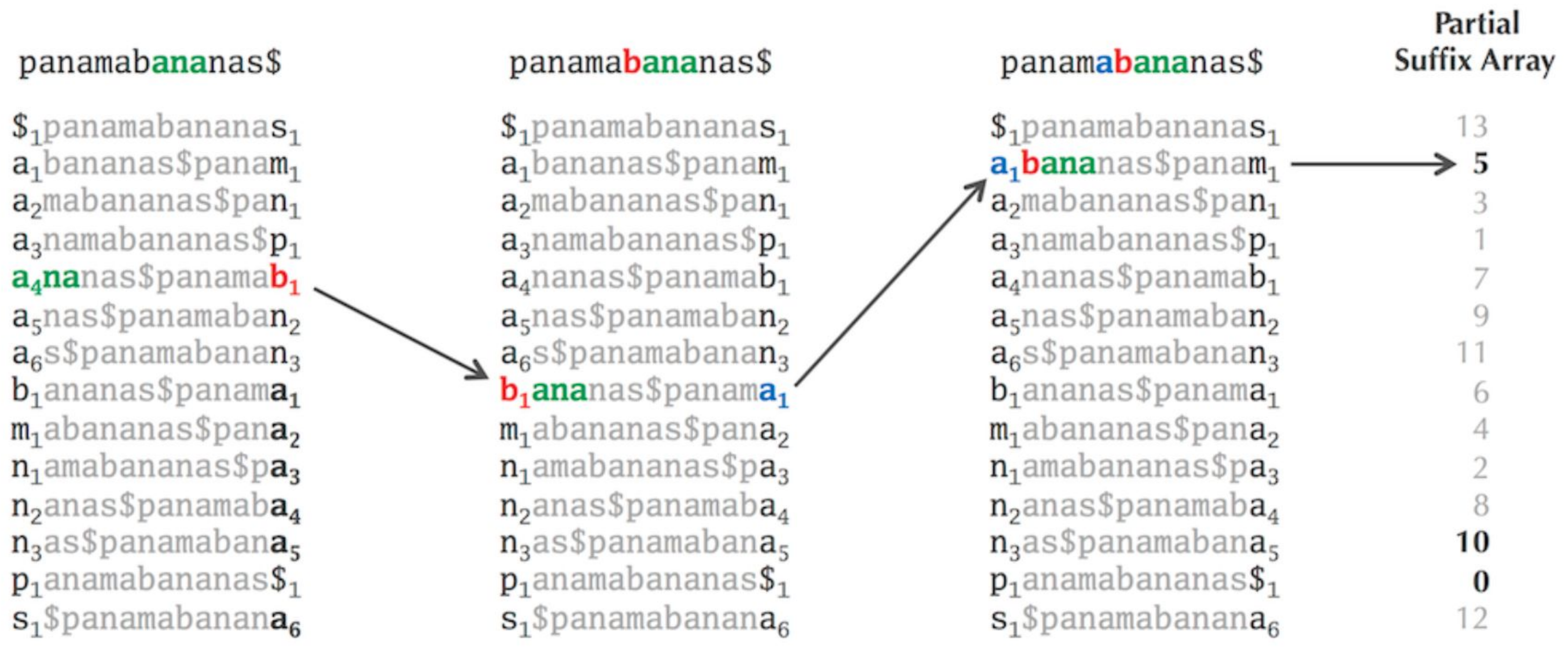
- Use suffix array.



M(*Text*)	SuffixArray(*Text*)
$ p a n a m a b a n a n a s	13
a b a n a n a s $ p a n a m	5
a m a b a n a n a s $ p a n	3
a n a m a b a n a n a s $ p	**1**
a n a n a s $ p a n a m a b	**7**
a n a s $ p a n a m a b a n	**9**
a s $ p a n a m a b a n a n	11
b a n a n a s $ p a n a m a	6
m a b a n a n a s $ p a n a	4
n a m a b a n a n a s $ p a	2
n a n a s $ p a n a m a b a	8
n a s $ p a n a m a b a n a	10
p a n a m a b a n a n a s $	0
s $ p a n a m a b a n a n a	12

Where are the matched patterns?

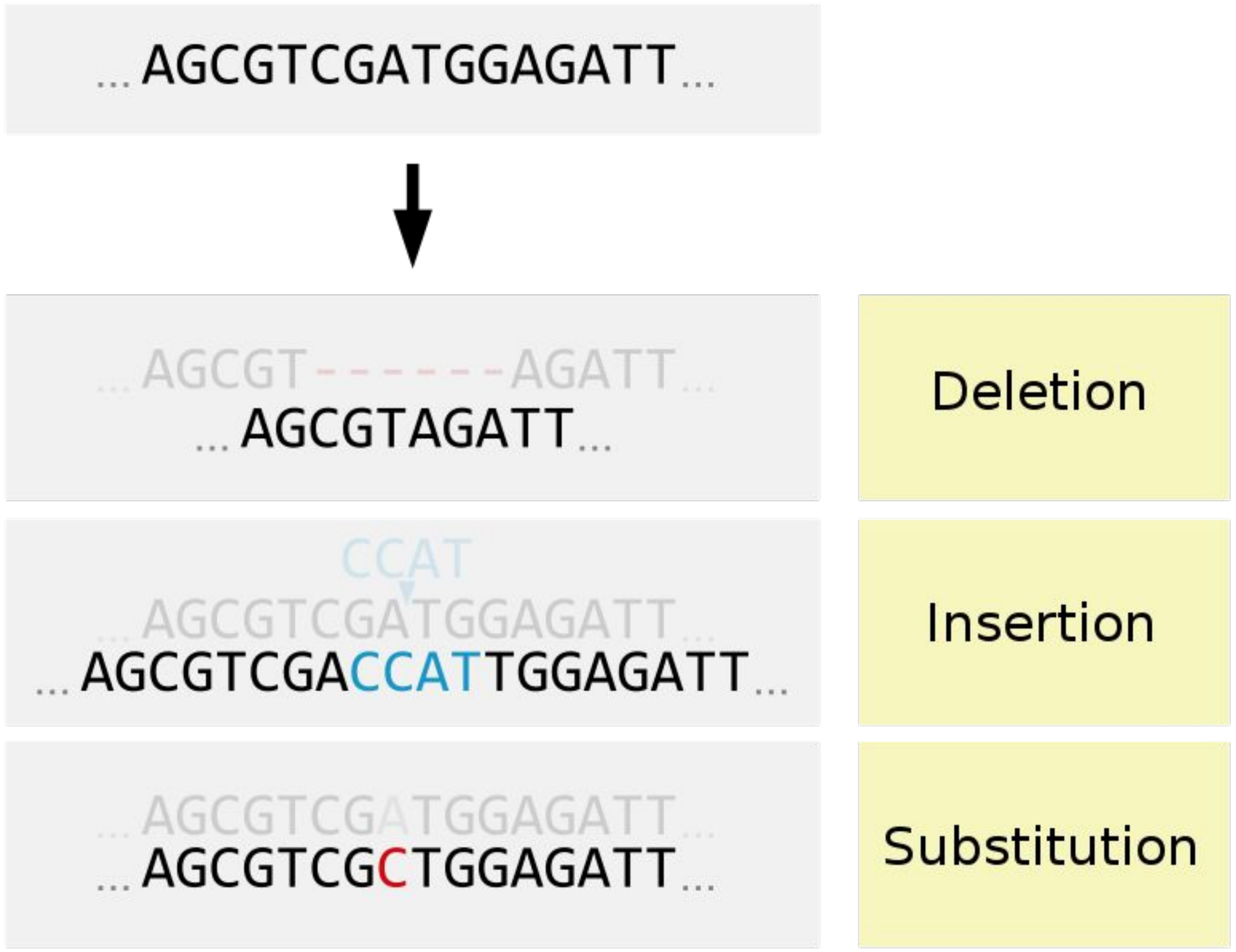
- Also use checkpoints to save memory.



Project 1
(created by Luke Li)

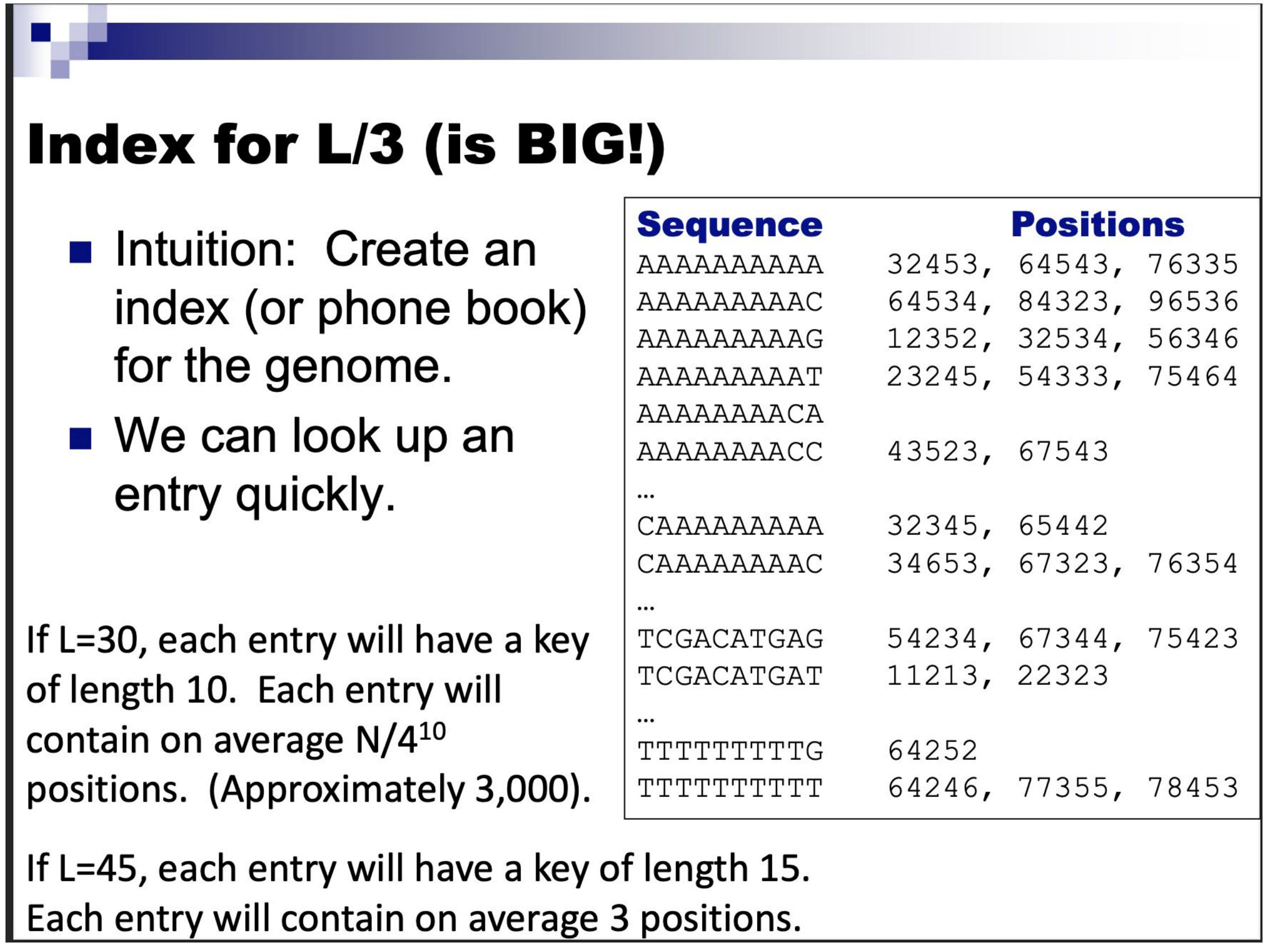
Types of genetic mutations used for grading

https://commons.wikimedia.org/wiki/File:Deletion_Insertion_Substitution-en.svg



What data structure should you use to represent the genome?

Lecture 3



Index for L/3 (is BIG!)

- Intuition: Create an index (or phone book) for the genome.
- We can look up an entry quickly.

Sequence	Positions
AAAAAAAAAA	32453, 64543, 76335
AAAAAAAAAC	64534, 84323, 96536
AAAAAAAAAG	12352, 32534, 56346
AAAAAAAAAT	23245, 54333, 75464
AAAAAAAACA	
AAAAAAAACC	43523, 67543
…	
CAAAAAAAAA	32345, 65442
CAAAAAAAAC	34653, 67323, 76354
…	
TCGACATGAG	54234, 67344, 75423
TCGACATGAT	11213, 22323
…	
TTTTTTTTTG	64252
TTTTTTTTTT	64246, 77355, 78453

If L=30, each entry will have a key of length 10. Each entry will contain on average $N/4^{10}$ positions. (Approximately 3,000).

If L=45, each entry will have a key of length 15. Each entry will contain on average 3 positions.

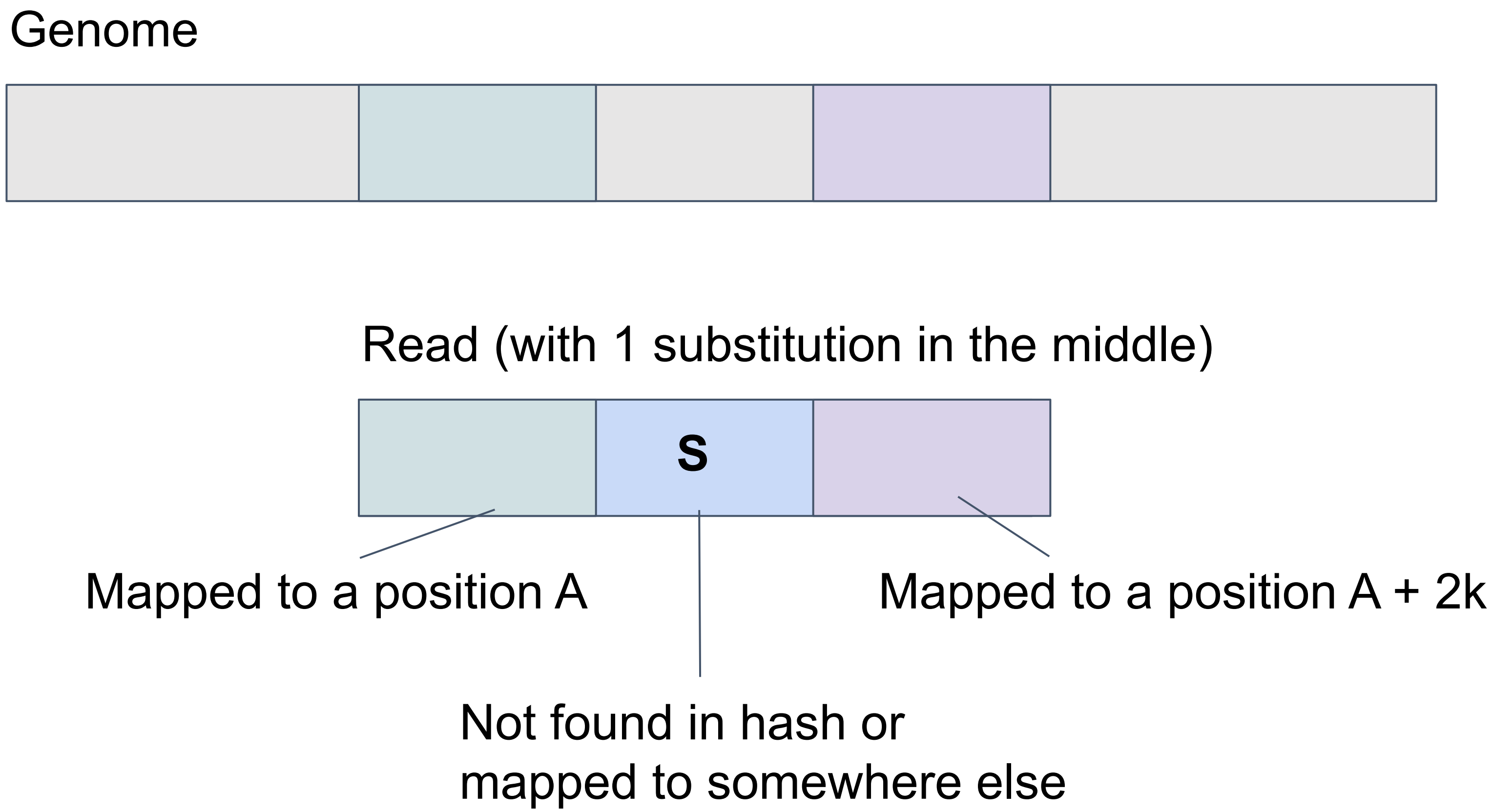
You can create a hash table containing positions of all unique k-mers in the genome.

You can also use tries or BWT.

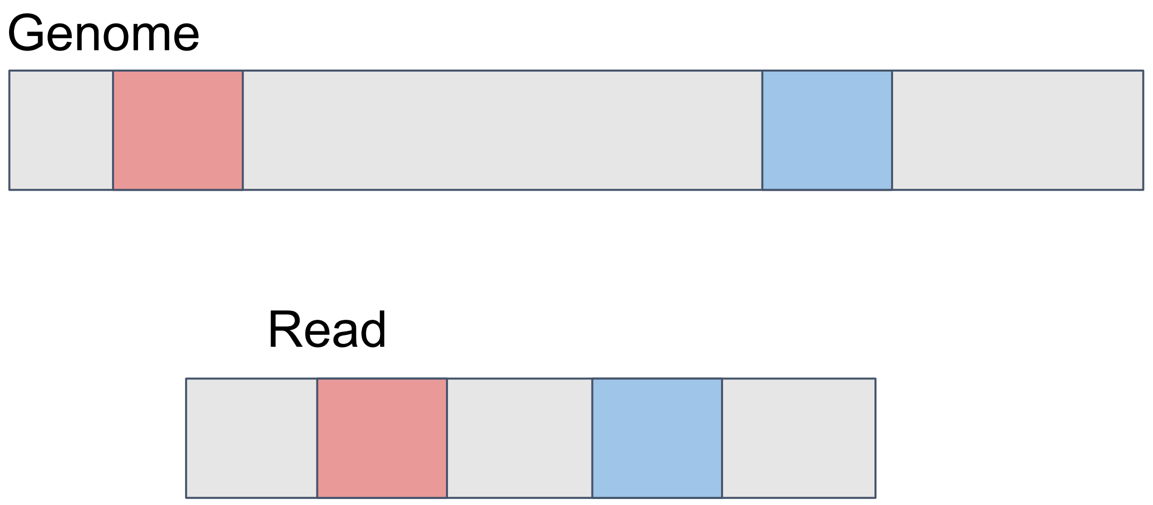
How do you map a read to the genome?

Assume you have a read of length L and a hash table k-mer positions.

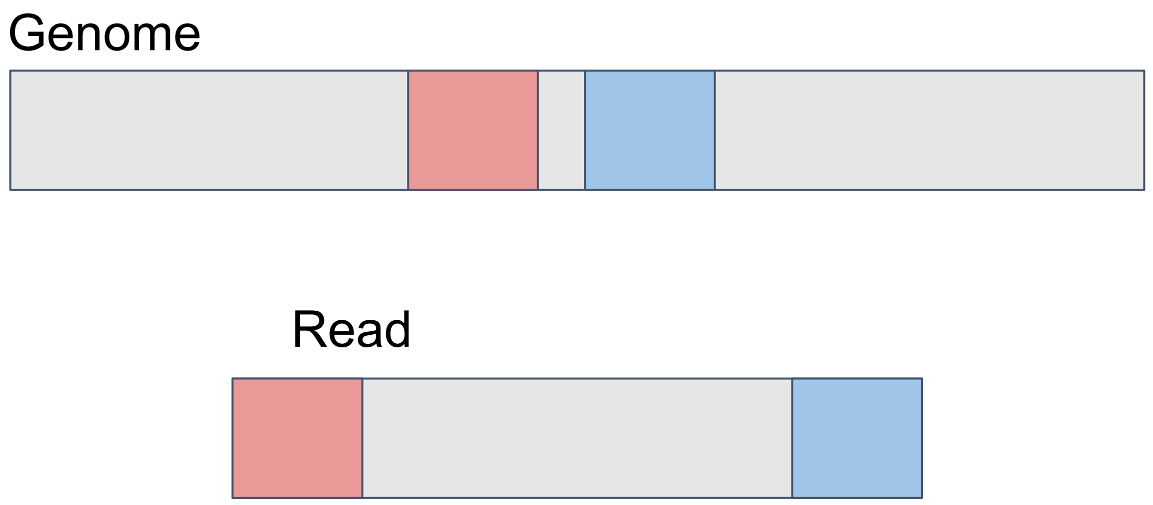
You can cut the read into L/k fragments and try to find the position of each fragment.


Genome
Read (with 1 substitution in the middle)
S
Mapped to a position A
Mapped to a position A + 2k
Not found in hash or mapped to somewhere else

How would indels look like on your read?



Deletion: some fragments of your read will map to positions that are more far apart on the genome



Insertion: some fragments of your read will map to positions that are closer on the genome