Short Read Alignment

Introduction to Bioinformatics Course

Niloufar Razani
Department of Computer Engineering
Sharif University of Technology

Fall 2023

Aligning Short Reads to the Reference Genome

- **Reference Genome:** database genome used for comparison.
- **Question:** how can we assemble individual genomes efficiently using the reference?

Why Not Use Assembly?

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

Read Mapping

- Determining where each read has high similarity to the reference genome.

CTGAGGATGGACTACGCTACTACTGATAGCTGTTT Reference

GAGGA CCACG TGA_A Reads

- For read mapping we could align each read to the reference genome to find the most similar region.

- This method is guaranteed to solve the problem of mapping reads to a reference genome, but its runtimes become a bottleneck when we scale to millions of reads.

Multiple Pattern Matching

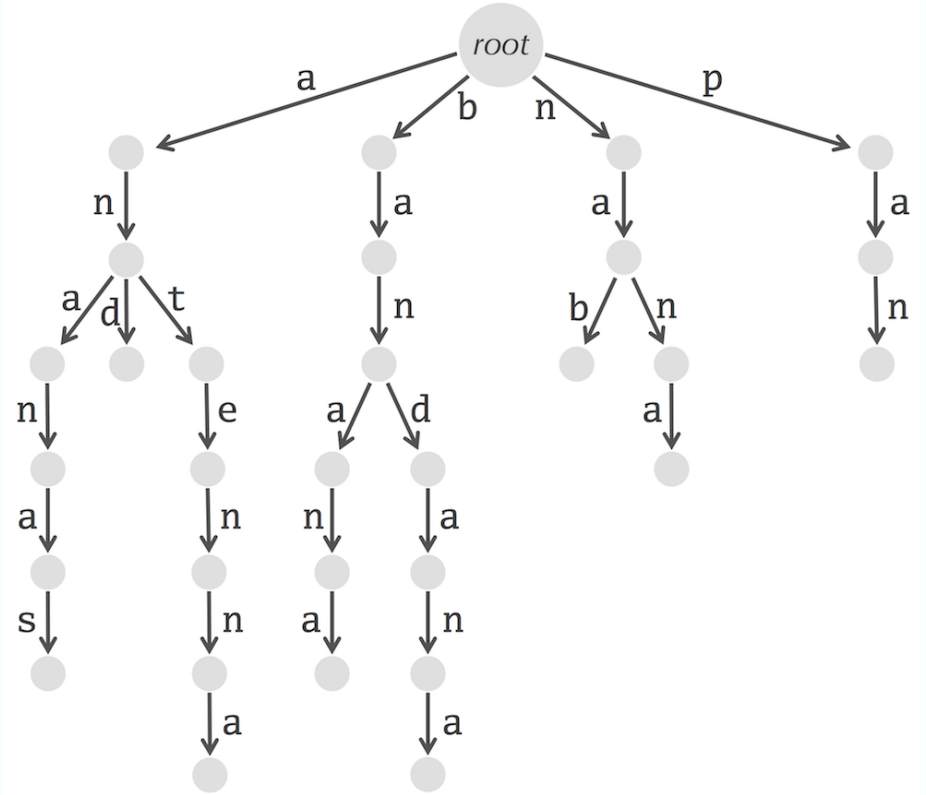
Multiple Pattern Matching Problem: Find all occurrences of a collection of patterns in a text.

- **Input:** A string *Text* and a collection *Patterns* containing (shorter) strings.
- **Output:** All starting positions in *Text* where a string from *Patterns* appears as a substring.

BruteForcePatternMatching

- Slide each Pattern along Text, checking whether the substring starting at each position of Text matches Pattern.
- Therefore, the total runtime of BruteForcePatternMatching for the Multiple Pattern Matching Problem is $O(|Text| \cdot |Patterns|)$. So it is too slow.

Herding Patterns into a Trie

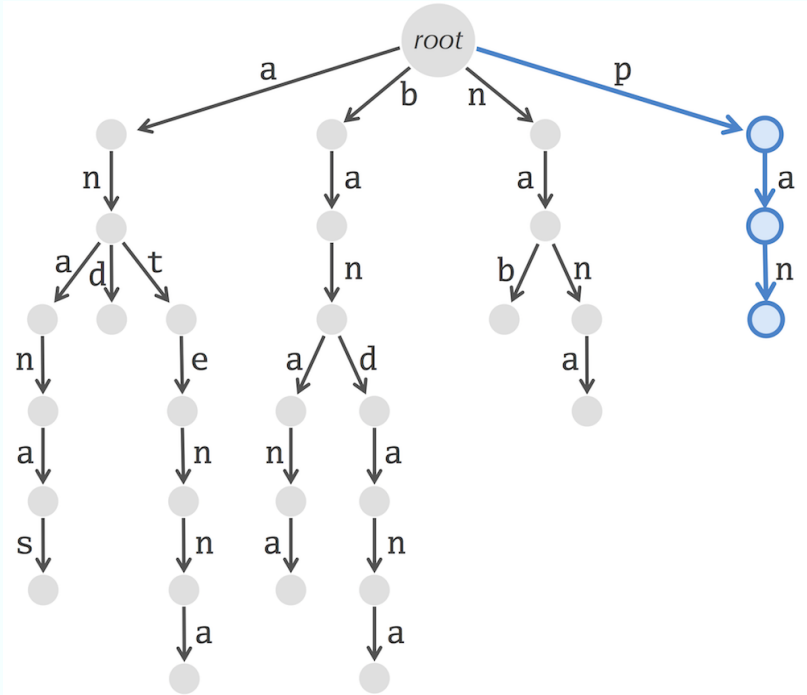


The trie for the following *Patterns*:
"ananas", "and", "antenna", "banana", "bandana", "nab", "nana", "pan".

The most obvious way to construct Trie(Patterns) is by iteratively adding each string from Patterns to the growing trie.

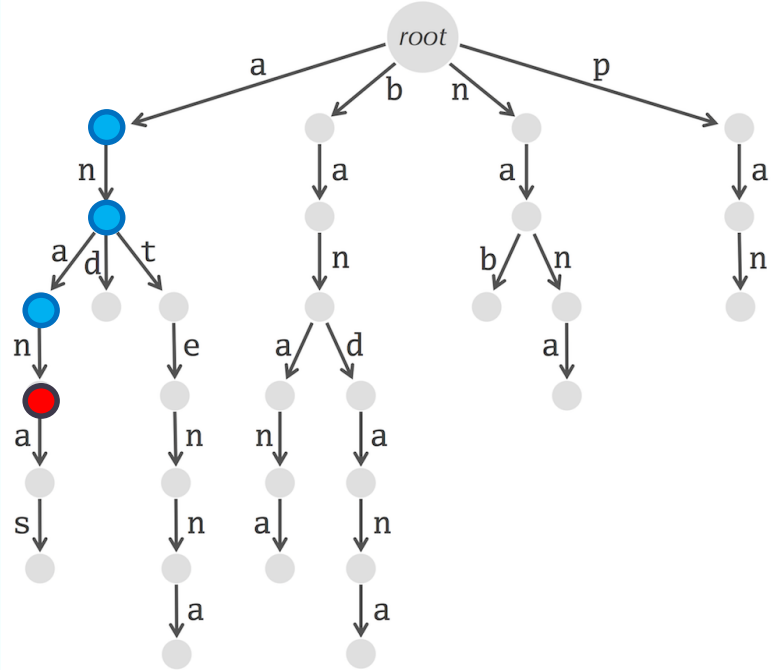
Applying the Trie to Multiple Pattern Matching

panamabananas



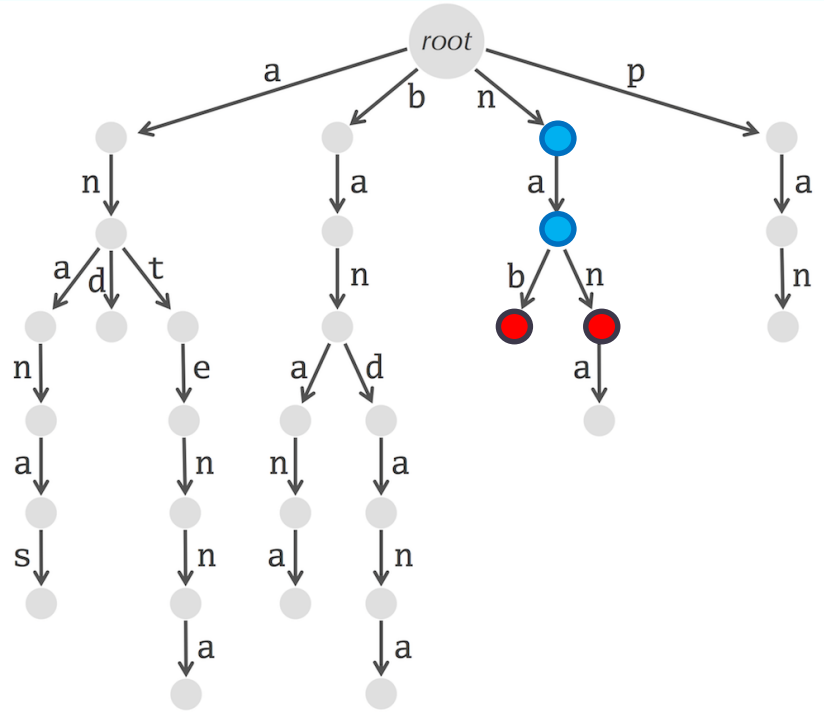
Applying the Trie to Multiple Pattern Matching

panamabananas



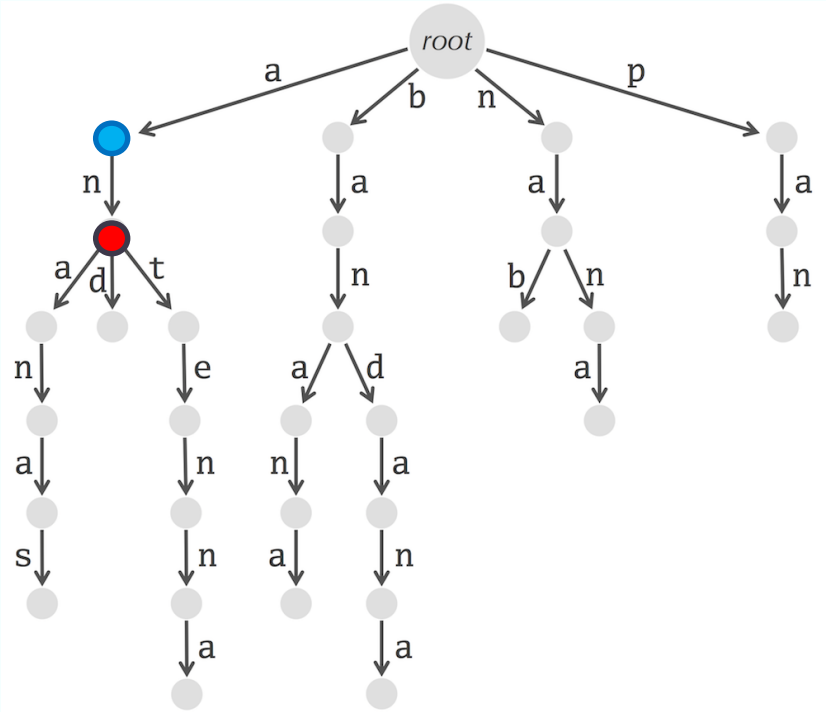
Applying the Trie to Multiple Pattern Matching

panamabananas



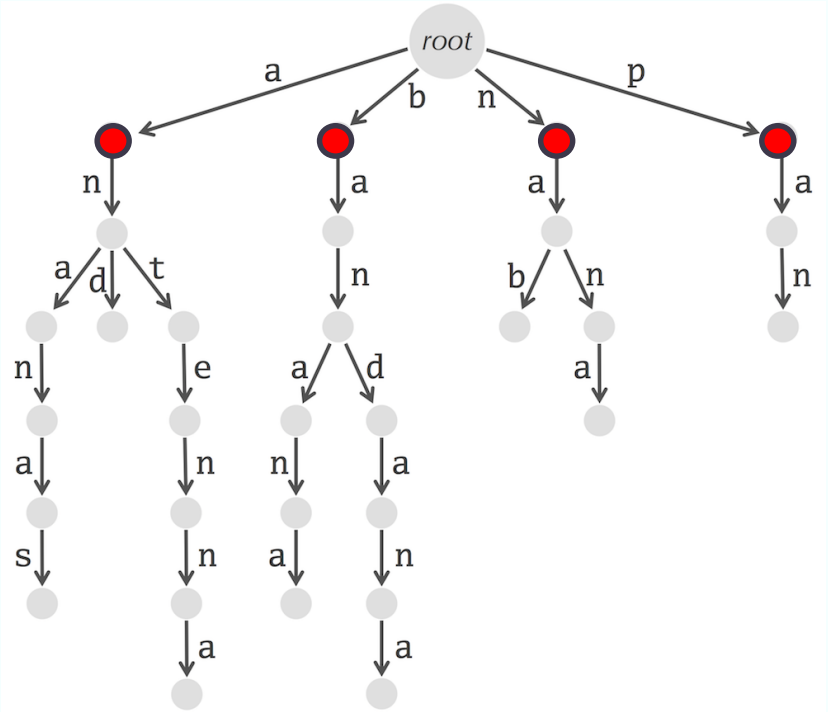
Applying the Trie to Multiple Pattern Matching

panamabananas



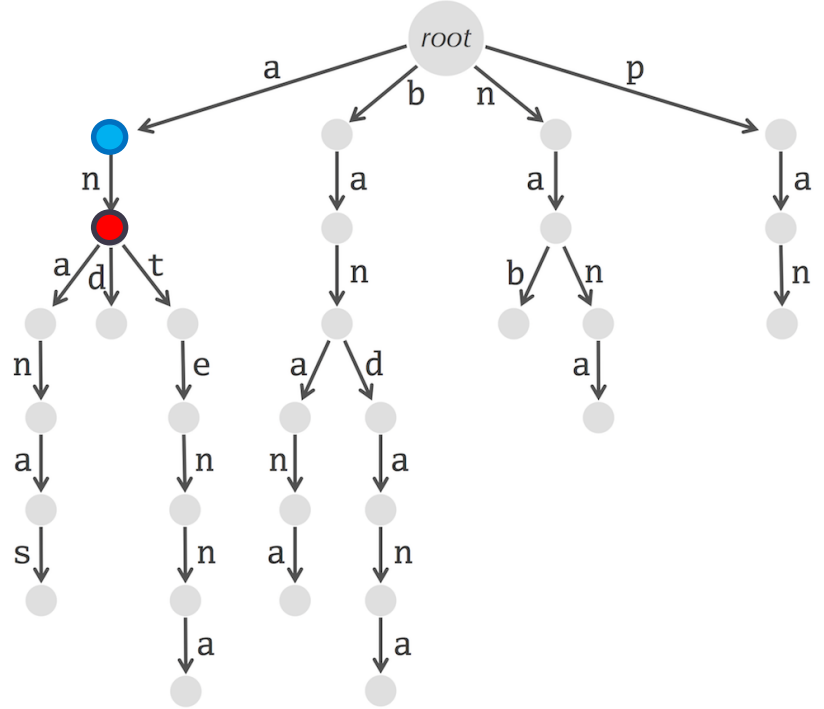
Applying the Trie to Multiple Pattern Matching

panamabananas



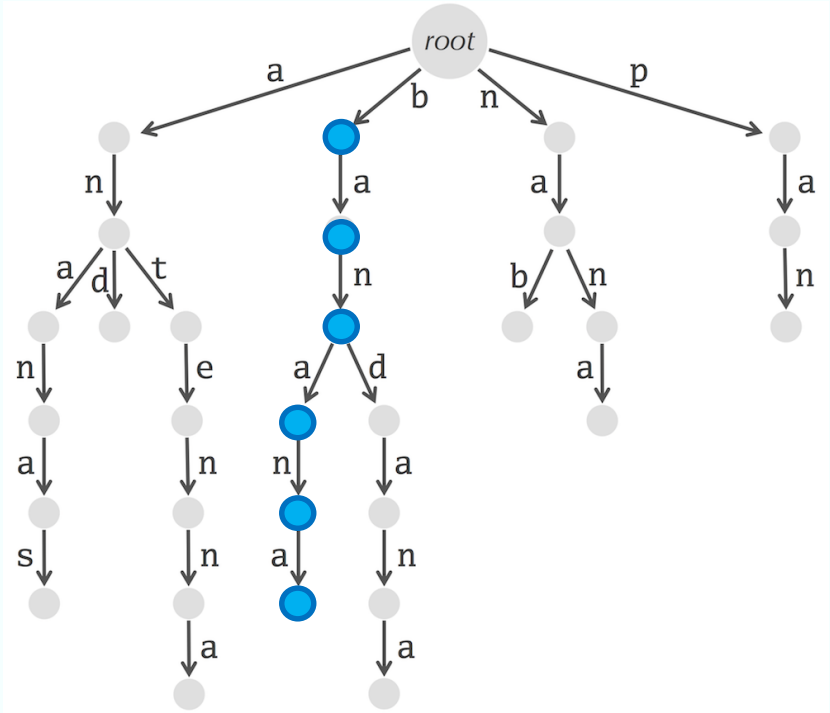
Applying the Trie to Multiple Pattern Matching

panamabananas



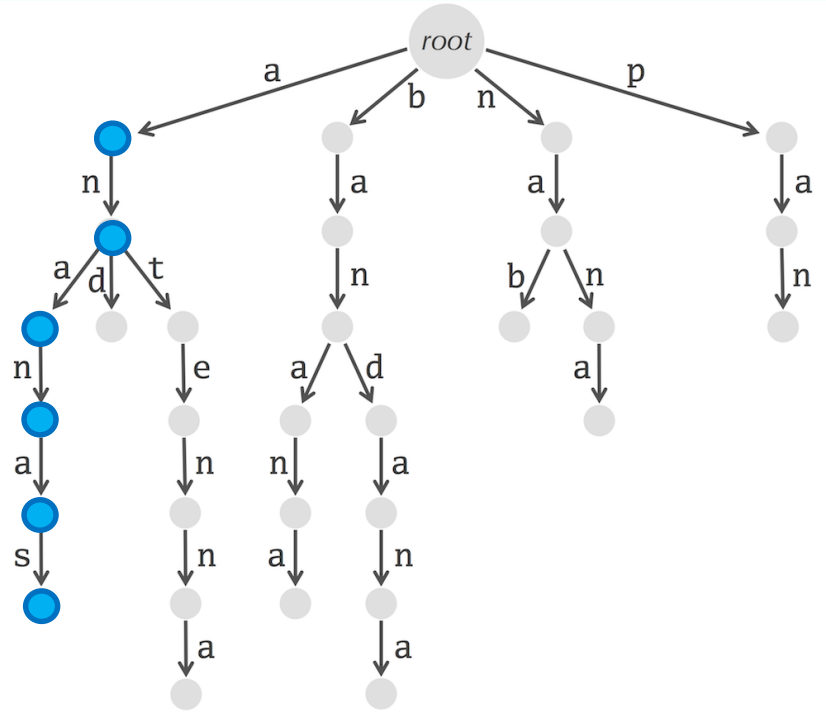
Applying the Trie to Multiple Pattern Matching

panamabananas



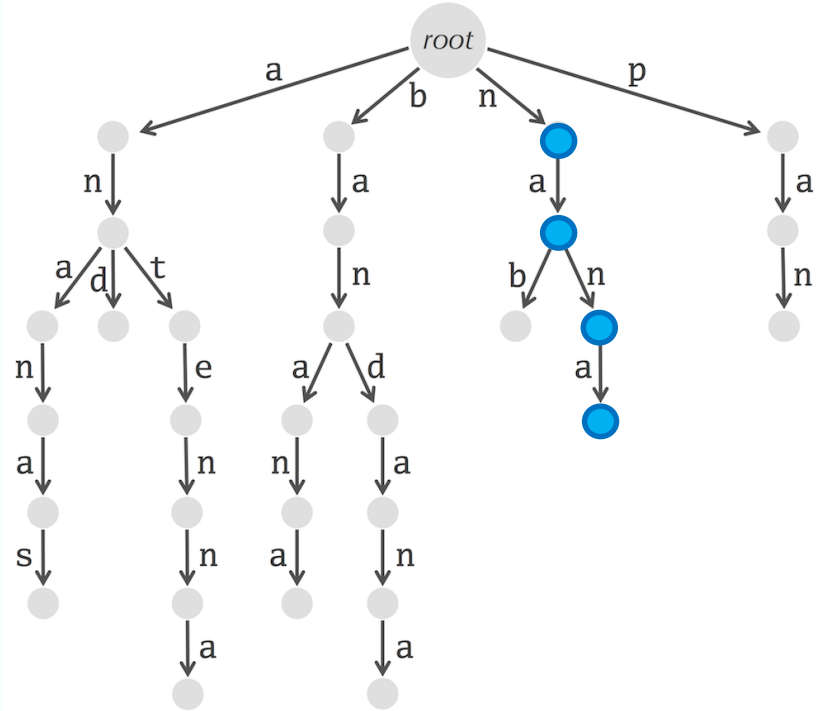
Applying the Trie to Multiple Pattern Matching

panamabananas



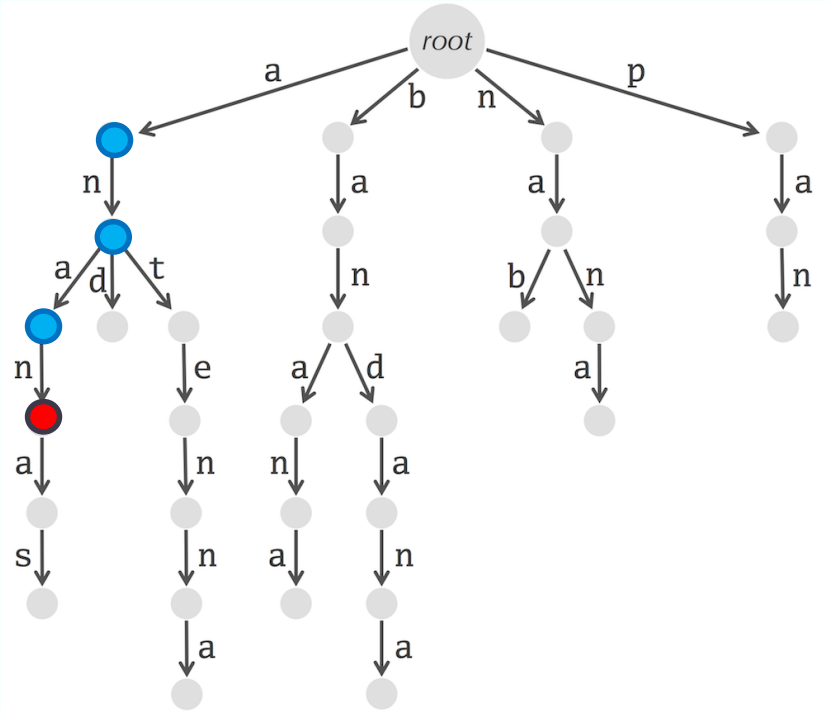
Applying the Trie to Multiple Pattern Matching

panamabananas



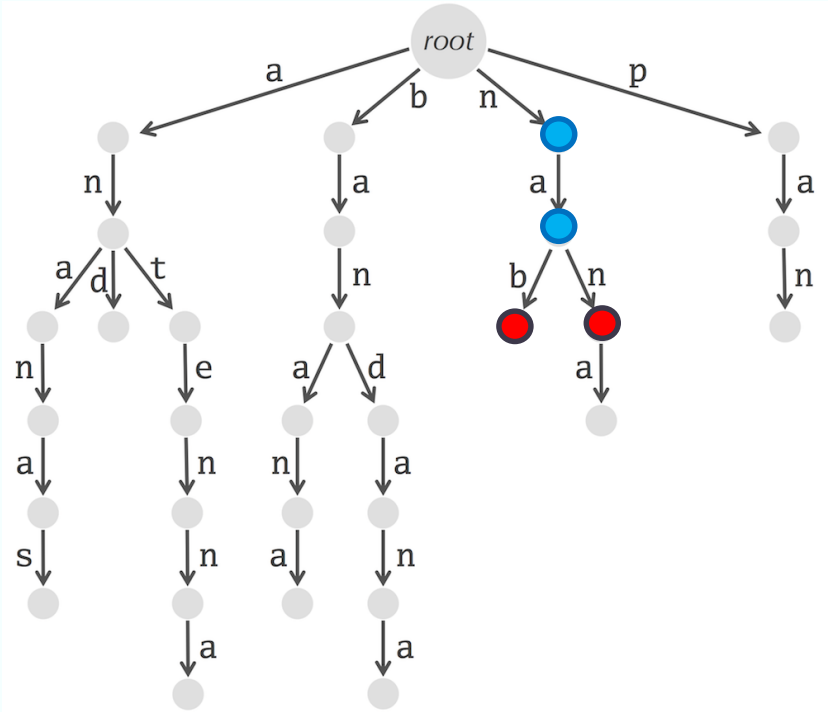
Applying the Trie to Multiple Pattern Matching

panamabananas



Applying the Trie to Multiple Pattern Matching

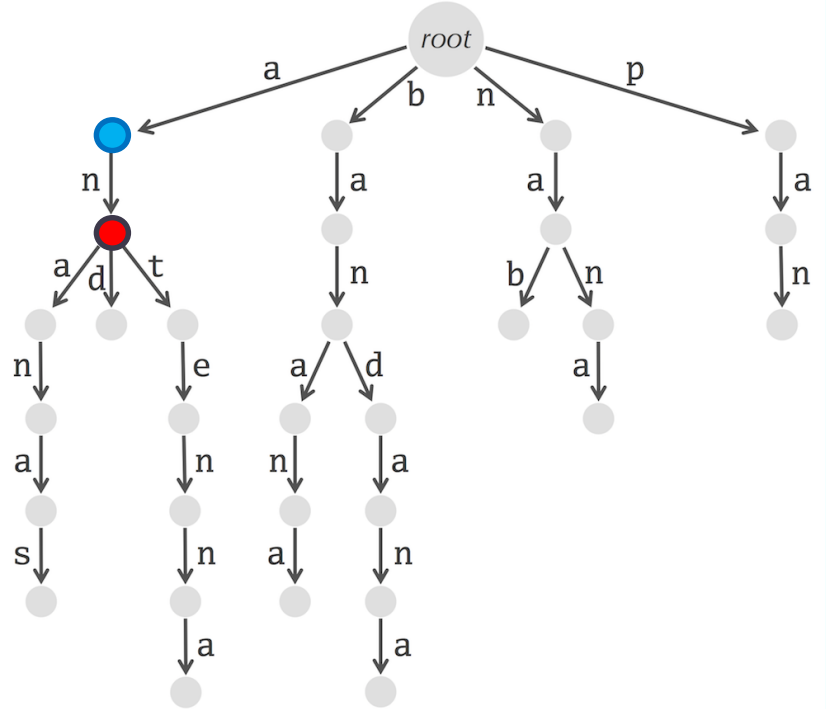
panamabananas



Applying the Trie to Multiple Pattern Matching

panamabananas

For **PrefixTrieMatching** to work, we have made a hidden assumption that no string in Patterns is a prefix of another string in Patterns



Applying the Trie to Multiple Pattern Matching

- The algorithm described on the previous step is called **PrefixTrieMatching**.
- **PrefixTrieMatching** finds whether any strings in *Patterns* match a prefix of *Text*.

```
TrieMatching(Text, Trie)
  while Text is nonempty
    PrefixTrieMatching(Text, Trie)
    remove first symbol from Text
```

Runtime of Brute Force:

Total : O(|genome| * |patterns|)

Runtime of Trie Maching:

Trie Construction : O(|patterns|)

Pattern Matching : O(|genome| * |longest pattern|)

- **Although TrieMatching is fast, storing a trie consumes a lot of memory.**

Worst Case: O(|patterns|)

Preprocessing the Genome

- What if instead we create a data structure from the genome itself?
- Our goal is to compare each string in *Patterns* against *Text* without needing to traverse *Text* from beginning to end.

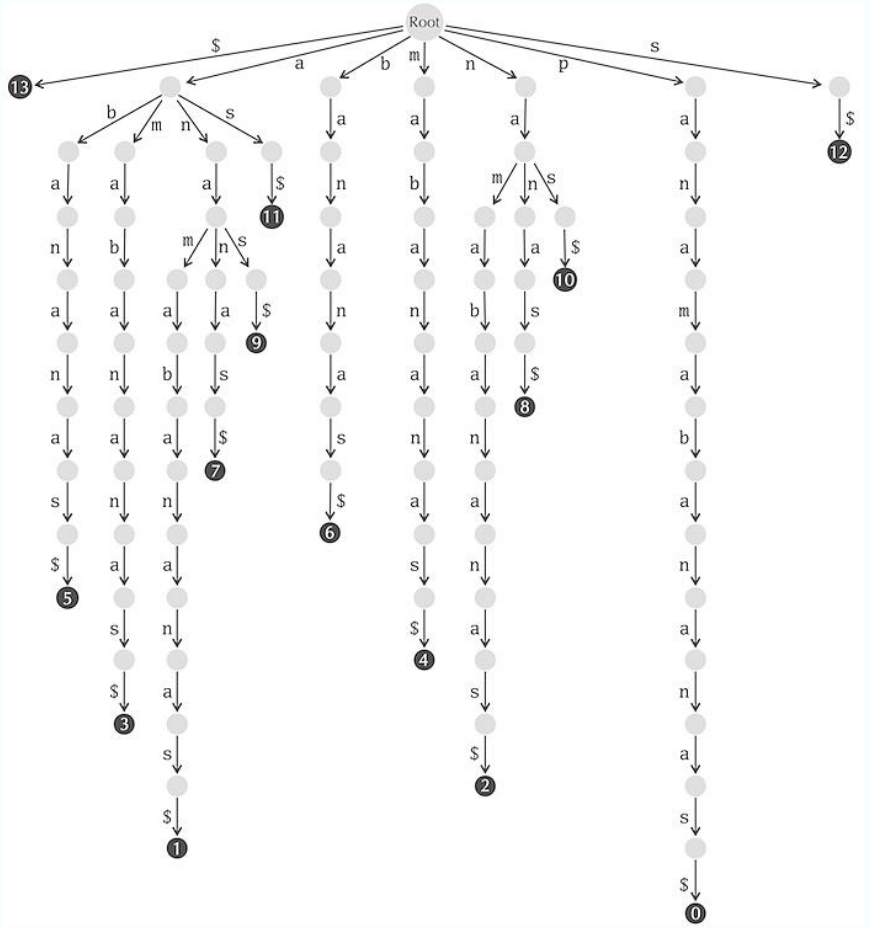
Form all suffixes of Genome.

How can we combine these suffixes into a data structure?

Let's use a Trie!

Suffix Tries

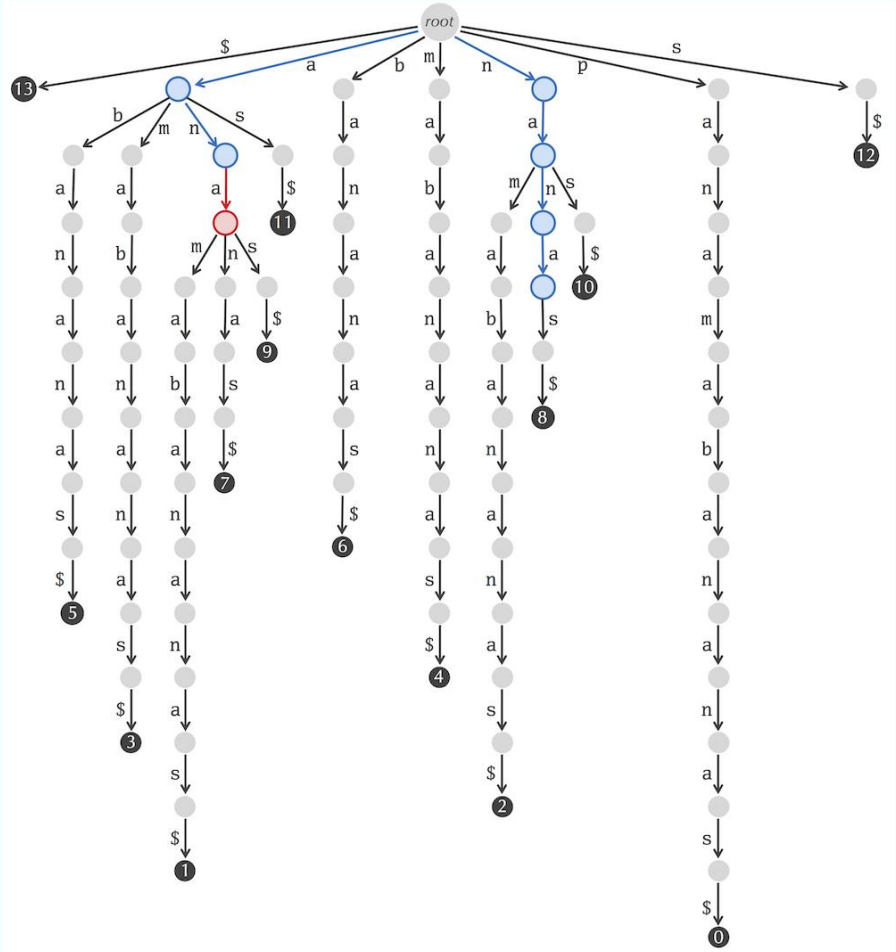
- A **suffix trie**, denoted **SuffixTrie(Text)**, is the trie formed from all suffixes of Text.
- We append the dollar-sign ("$") to Text in order to mark the end of Text.
- We will also label each leaf of the resulting trie by the **starting position of the suffix** whose path through the trie ends at this leaf.



SuffixTrie("panamabananas$")

Using Suffix Tries for Pattern Matching

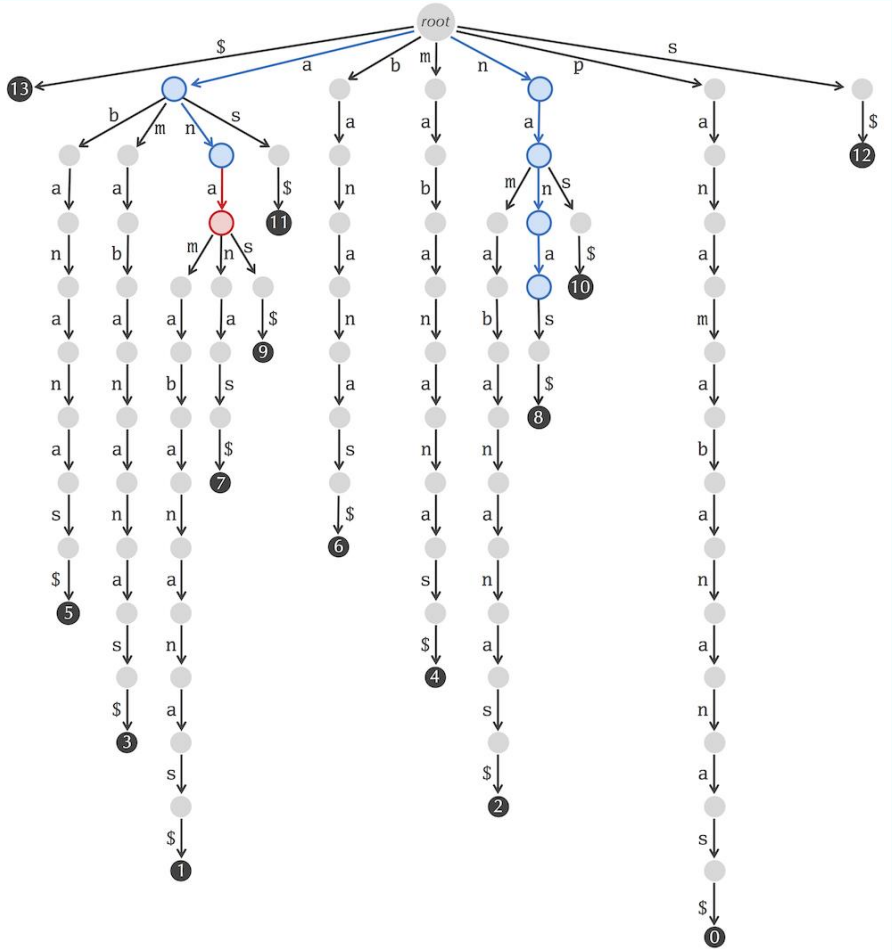
- We can determine whether Pattern occurs in SuffixTrie(Text) by starting at the root and spelling symbols of Pattern downward.
- If we can find a path in the suffix trie spelling out Pattern, then we know that Pattern must occur in Text.
- Threading "**antenna**" through SuffixTrie("panamabananas$") fails to find a match because no suffix of "panamabananas$" begins with "**ant**"; however, threading "**nanas**" through the suffix trie does find a match: "panamaba**nanas$**".



Using Suffix Tries for Pattern Matching

- This suffix trie illustrates how to find "**nanas**" in "**panamabananas$**", but it does not tell us where "nanas" occurs in Text. How can we obtain this information?

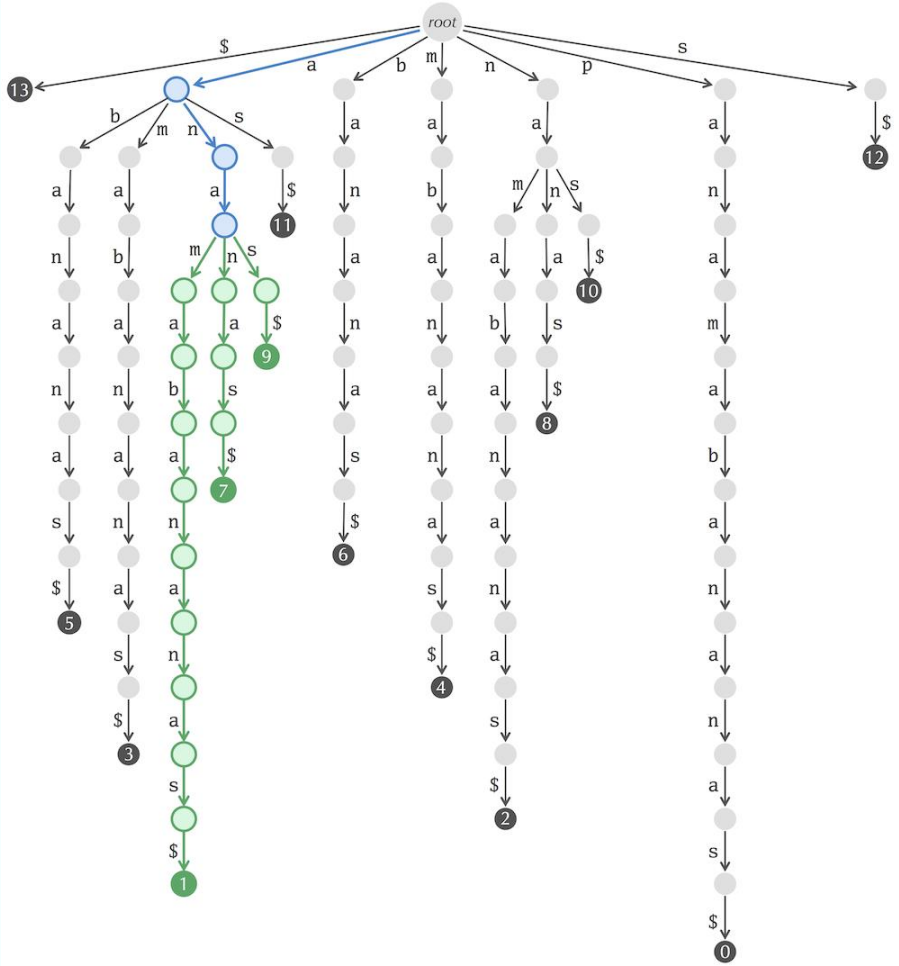
- We can consult **the label at that leaf to determine the starting position of the suffix.**



Using Suffix Tries for Pattern Matching

- If the path spelling out Pattern stops before a leaf at some node v of SuffixTrie(Text), then Pattern may occur more than once in Text.
- To locate these matches, follow all paths from v down to the leaves of SuffixTrie(Text), which will indicate all starting positions of Pattern in Text.
- "ana" corresponds to a path in SuffixTrie("panamabananas$") ending at an internal node.

panamabananas$", "panamabananas$", and "panamabananas$



Memory Trouble Once Again

- **Worst Case: the suffix Trie holds O(|suffixes|) nodes.**

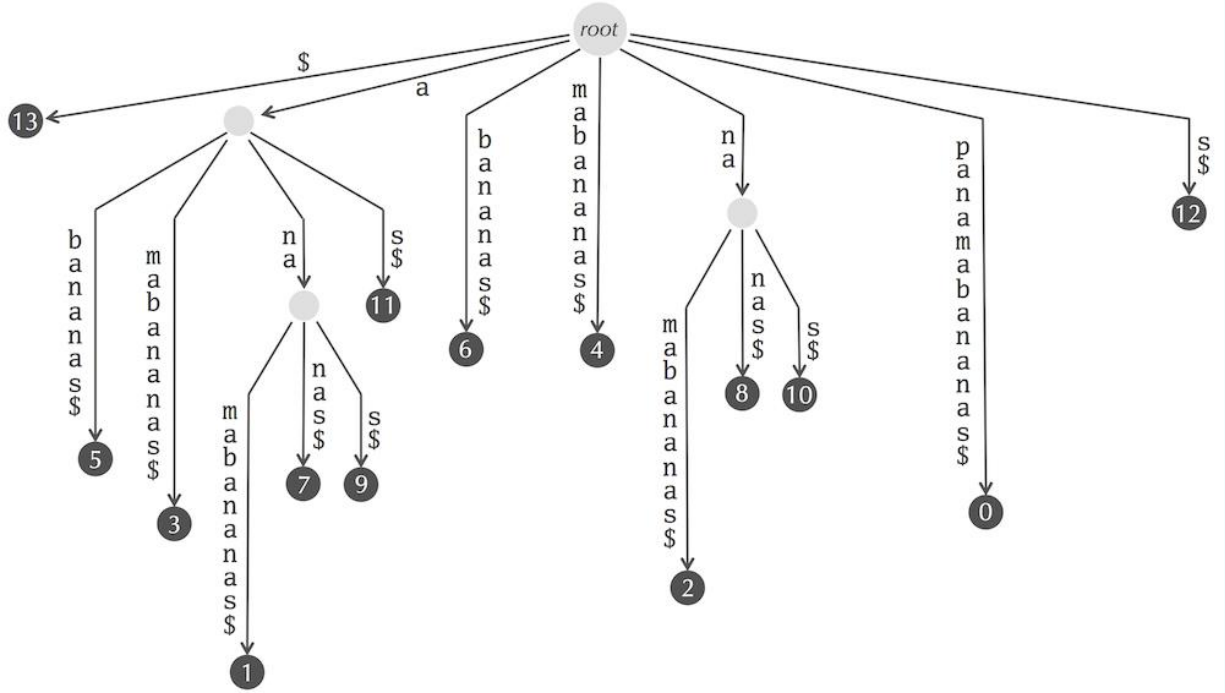
- **For a genome of length n, |suffixes| = n(n-1)/2 = O(n^2)**

Suffixes

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

Compressing the Trie

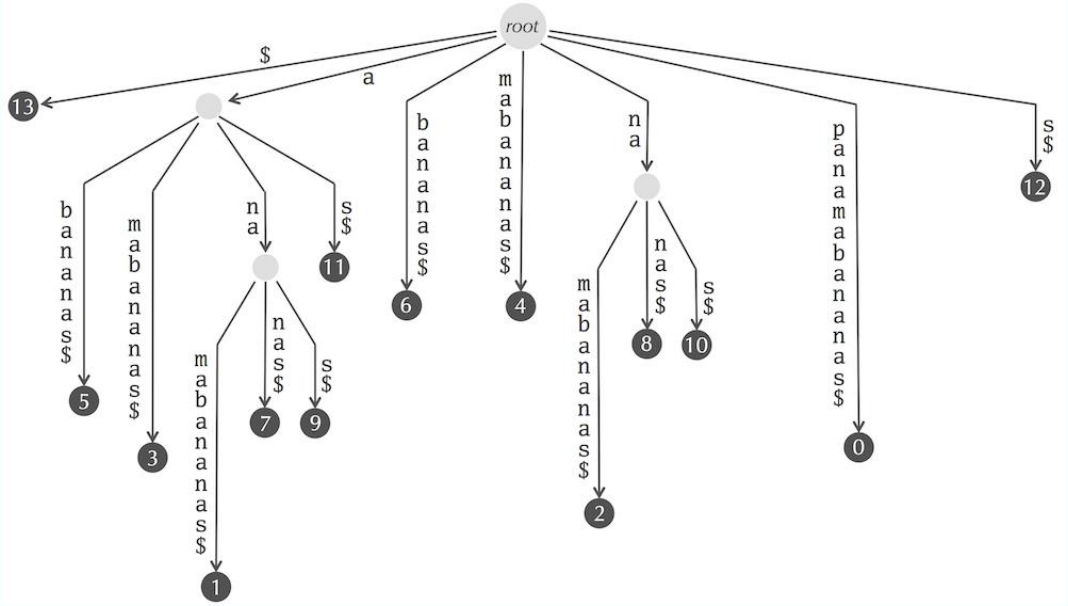
- To reduce memory, we can compress each **non-branching path** of the tree into an edge.
- This data structure is called a **suffix tree**.



Compressing the Trie

For any Genome, # nodes < 2 |Genome|.
leaves = |Genome|
internal nodes < |Genome|-1

We moved from quadratic memory usage to something with linear memory usage. About 20 * |Genome|



Genome compression

- Can we be so ambitious as to look for a data structure that would encode Text using memory approximately equal to the length of Text while still enabling fast pattern matching?

Idea #1 : Run Length Encoding
Compress a Run of n Identical symbol

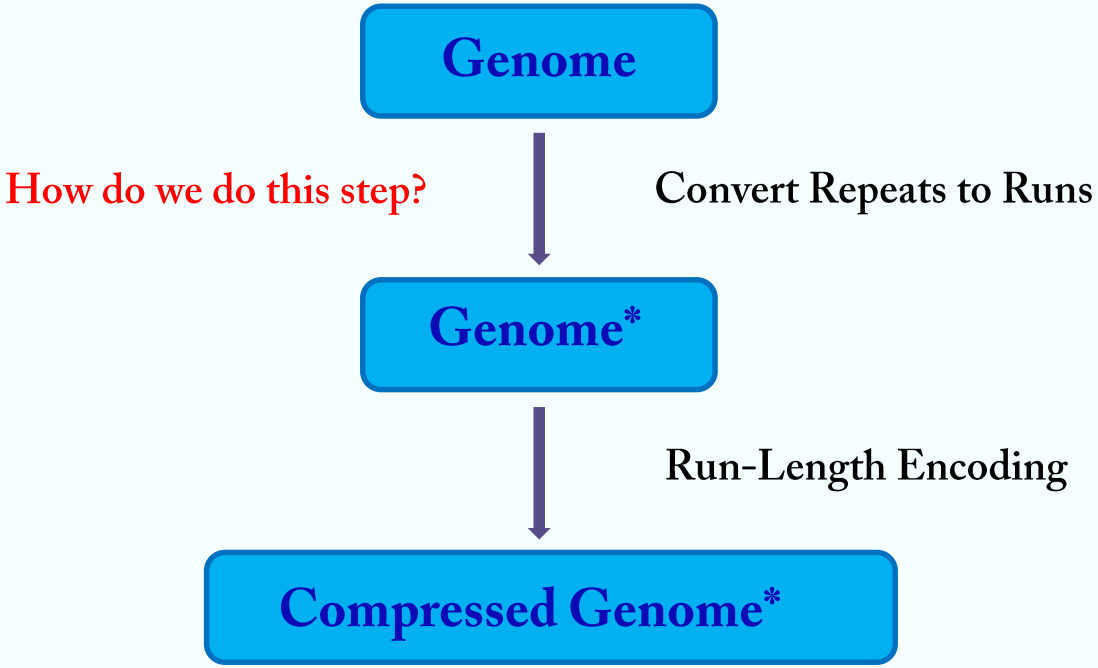
GGGGGGGGGGCCCCCCCCCCCAAAAAAATTTTTTTTTTTTTTTTCCCCCG

10G11C7A15T5C1G

Problem:

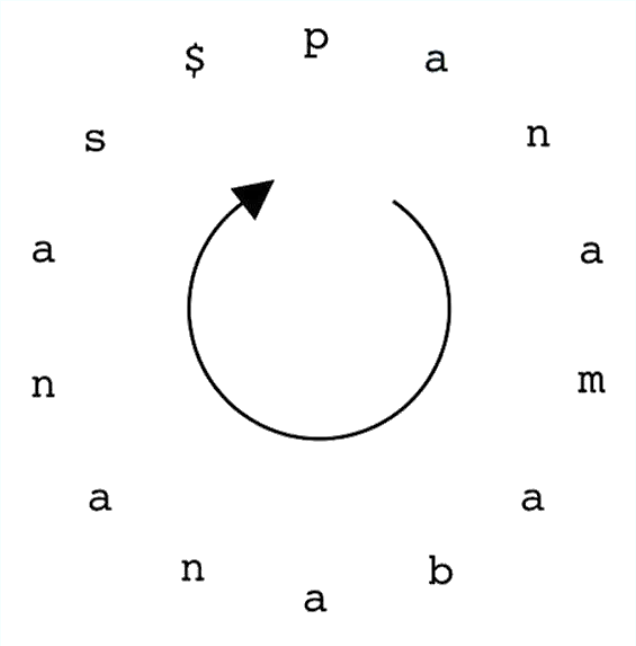
Genome don't have lots of runs
But they do have lots of repeats.

Converting Repeats to Runs



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$

The Burrows-Wheeler Transform

BWT Matrix

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

Sort the strings lexicographically

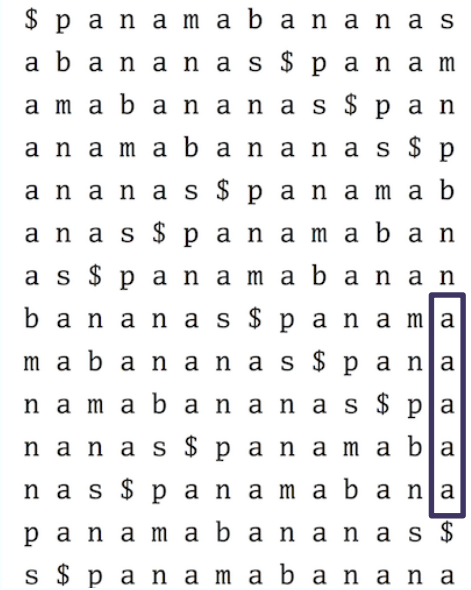
Burrows-Wheeler Transform

BWT("panamabananas$$") = "smnpbnnaaaaa$a"

Why Should we care?

```
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
```

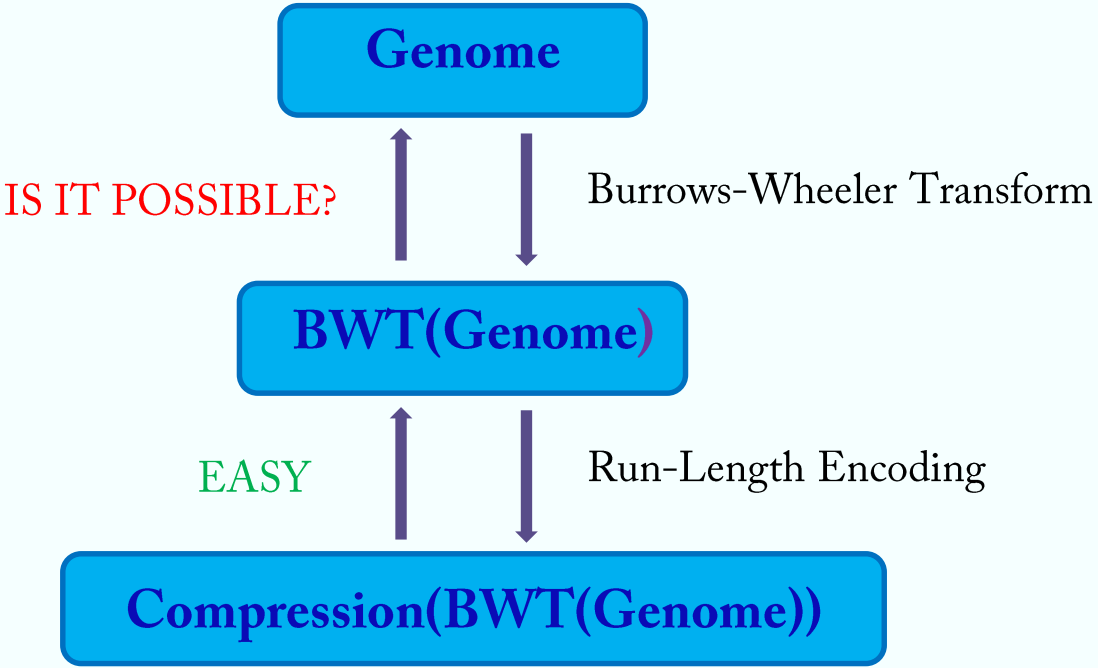
When all the cyclic rotations of Text(Watson and Crick's 1953 paper) are sorted lexicographically to form M(Text), all rows that begin with "nd..." and end with "...a" will tend to clump together.



The substring "ana" in "panamabananas$" plays the role of "and" in Watson and Crick's paper

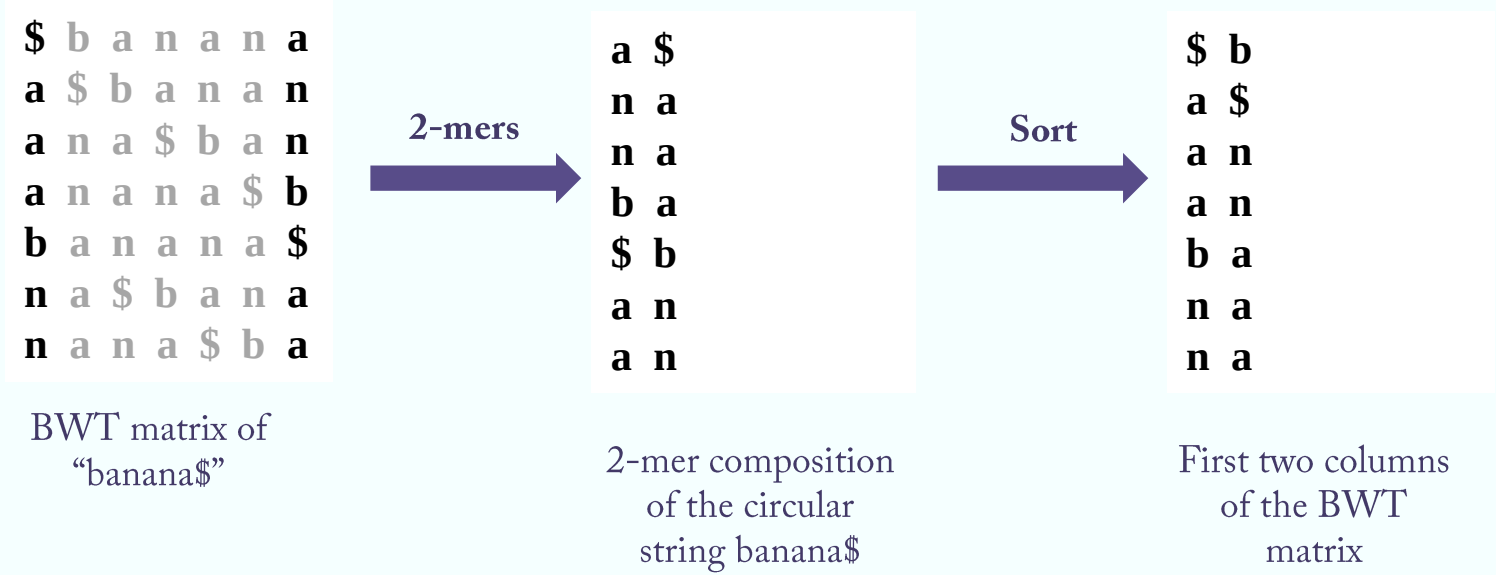
How Can We Decompress?

Why we chose the last column of BWT matrix and how can we decompress text from its BWT?

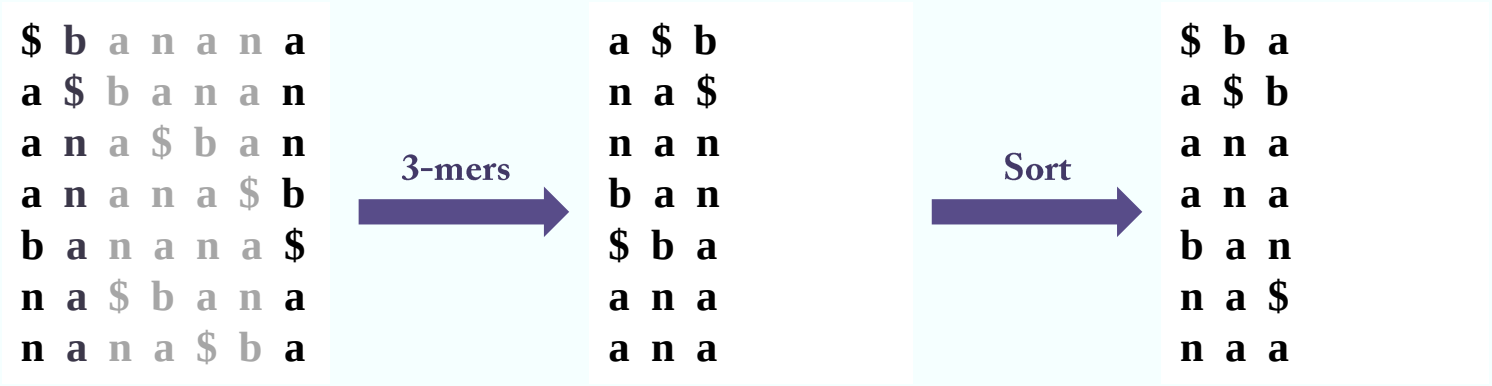


Reconstructing Genome from BWT

- Sorting all elements of last columns (BWT) gives first column of matrix. So we can get another column for free.



Reconstructing Genome from BWT

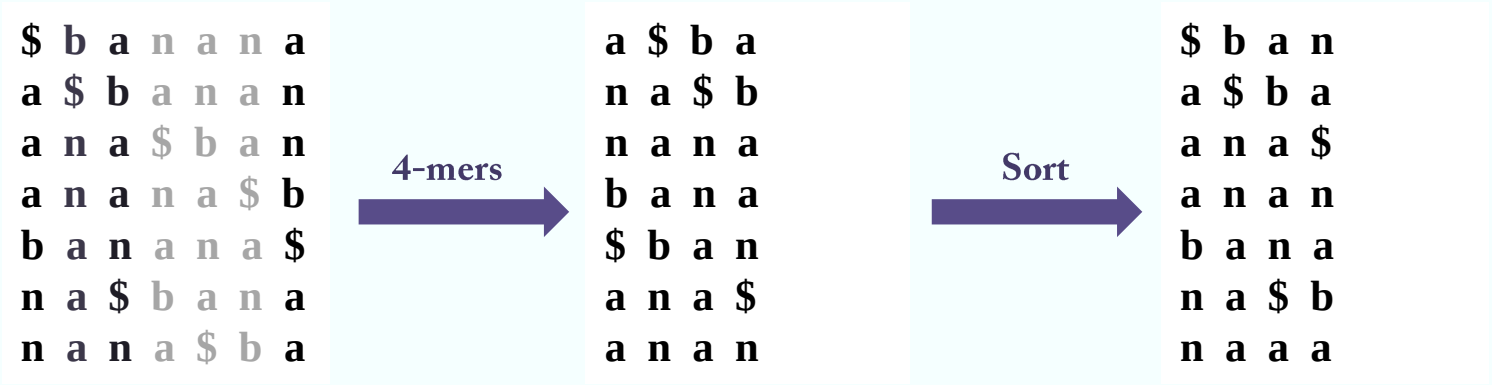


BWT matrix of "banana$"

3-mer composition of the circular string banana$

First three columns of the BWT matrix

Reconstructing Genome from BWT

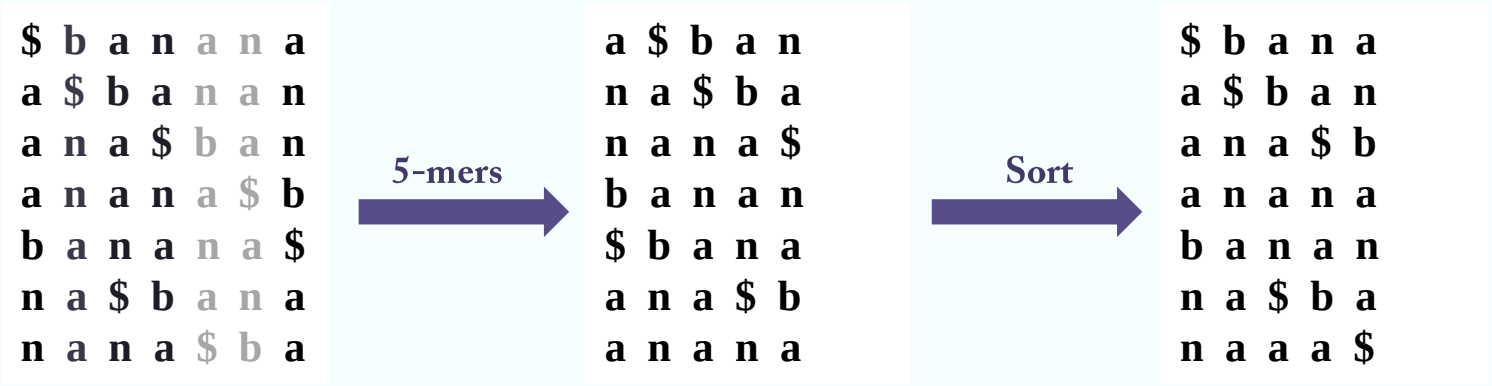


BWT matrix of "banana$"

4-mer composition of the circular string banana$

First four columns of the BWT matrix

Reconstructing Genome from BWT

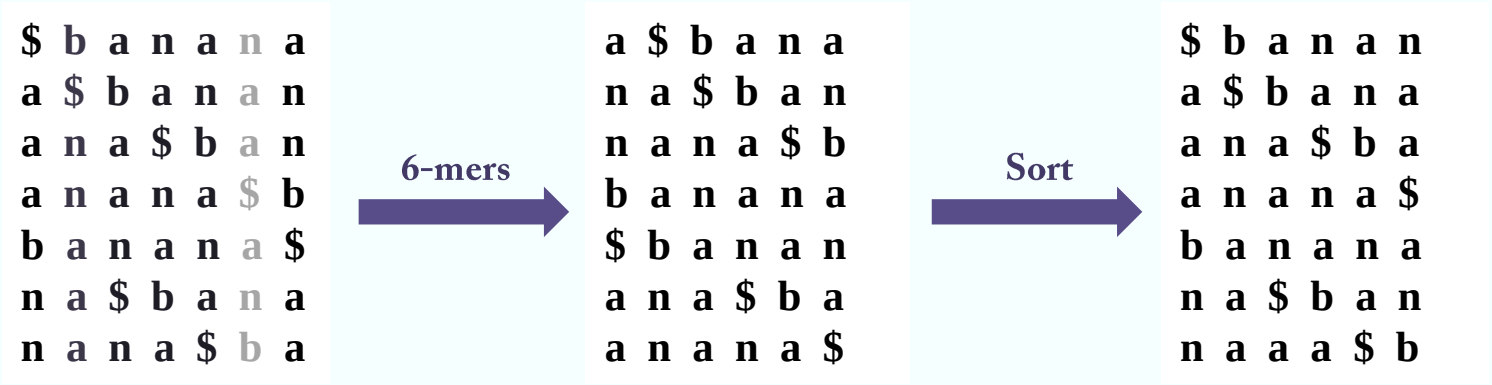


BWT matrix of "banana$"

5-mer composition of the circular string banana$

First five columns of the BWT matrix

Reconstructing Genome from BWT



BWT matrix of "banana$"

6-mer composition of the circular string banana$

First six columns of the BWT matrix

Reconstructing Genome from BWT

Now we know the entire matrix!

$	b	a	n	a	n	a
a	$	b	a	n	a	n
a	n	a	$	b	a	n
a	n	a	n	a	$	b
b	a	n	a	n	a	$
n	a	$	b	a	n	a
n	a	n	a	$	b	a

Taking all elements in the first row (after $) produce “banana”

More Memory Issues

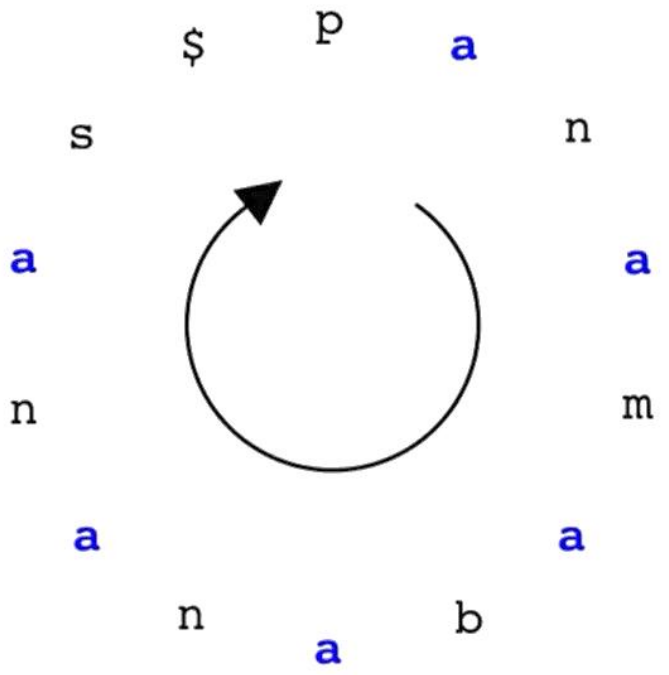
Reconstructing Genome from BWT(Genome) requires us to store |Genome| copies of |Genome|.

$	**b**	**a**	**n**	**a**	**n**	**a**
a	**$**	**b**	**a**	**n**	**a**	**n**
a	**n**	**a**	**$**	**b**	**a**	**n**
a	**n**	**a**	**n**	**a**	**$**	**b**
b	**a**	**n**	**a**	**n**	**a**	**$**
n	**a**	**$**	**b**	**a**	**n**	**a**
n	**a**	**n**	**a**	**$**	**b**	**a**

Can we invert BWT with less space?

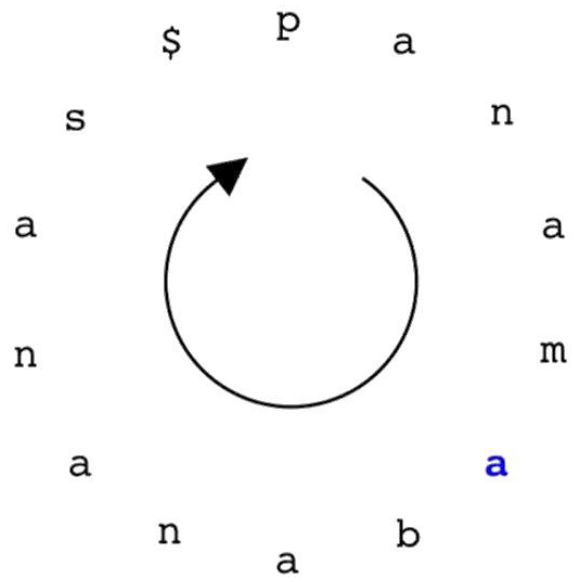
An Interesting Observation!

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



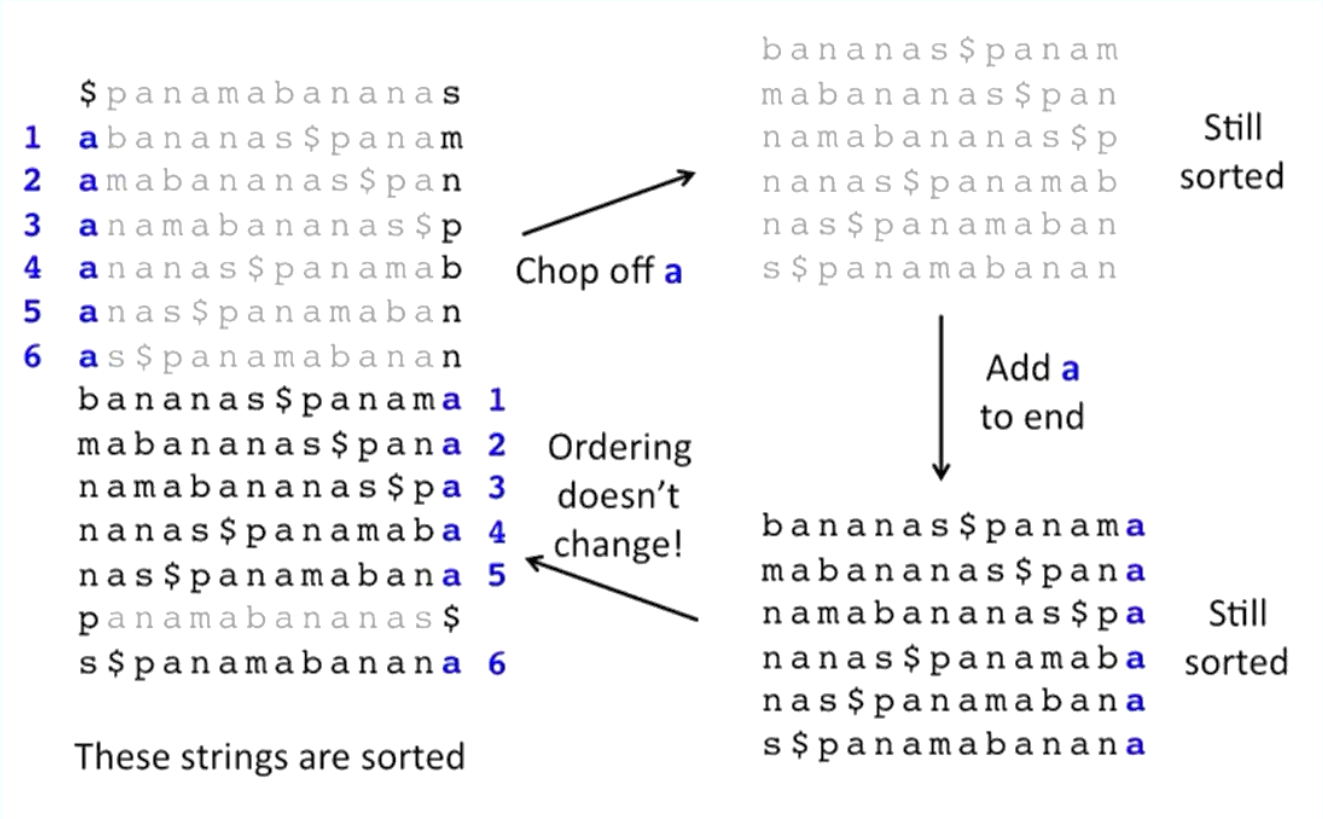
An Interesting Observation!

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



If we look at the first "a" in the first and last columns of BWT matrix, we notice that these two "a" correspond to the same position in the "**panamabananas$**" string.

Is It True in General?



$pa_3na_2ma_1ba_4na_5na_6s\$$

Is It True in General?

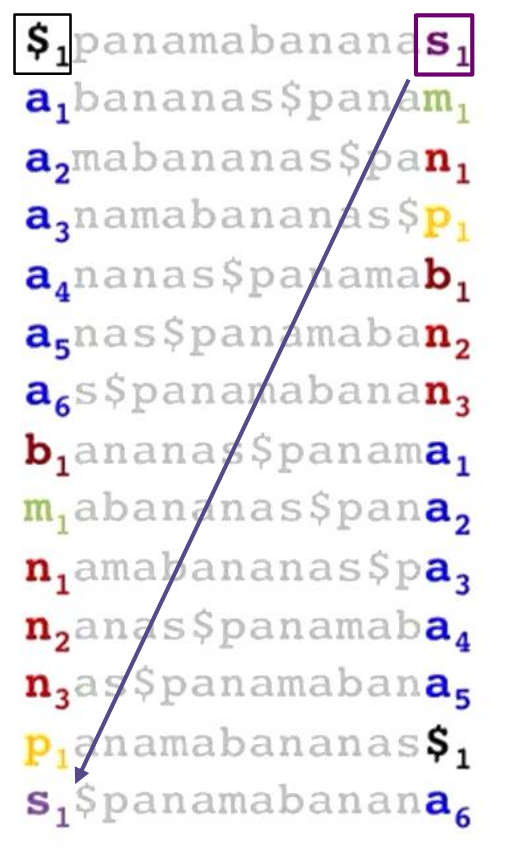
First-Last Property:

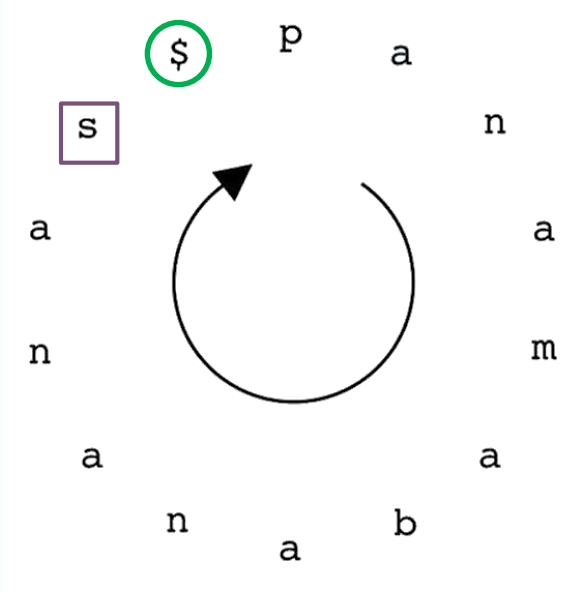
The k-th occurrence of symbol in first column and the k-th occurrence of symbol in last column correspond to the same position of symbol in Genome.

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

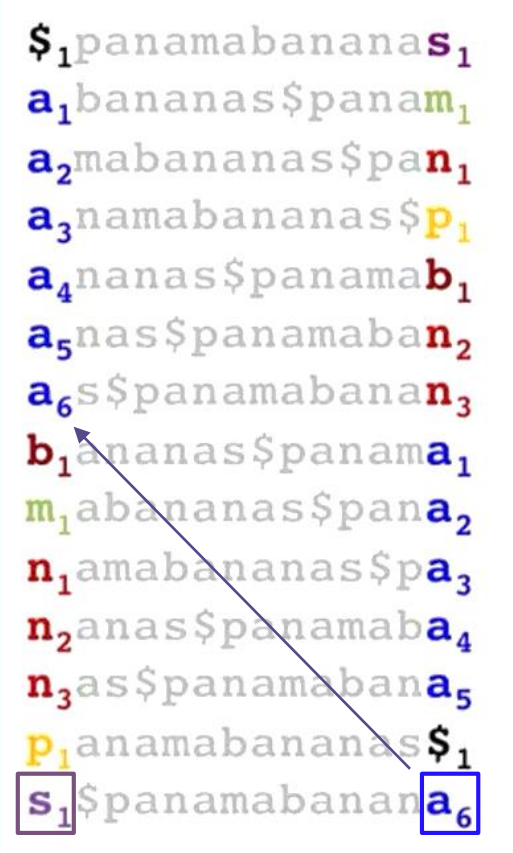
$\$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$$_1$
a$_4$nanas$panamab$$_1$
a$_5$nas$panamaban$$_2$
a$_6$s$panamabanan$$_3$
b$_1$ananas$panama$$_1$
m$_1$abananas$pana$$_2$
n$_1$amabananas$pa$$_3$
n$_2$anas$panamaba$$_4$
n$_3$as$panamabana$$_5$
p$_1$anamabananas$\$_1$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

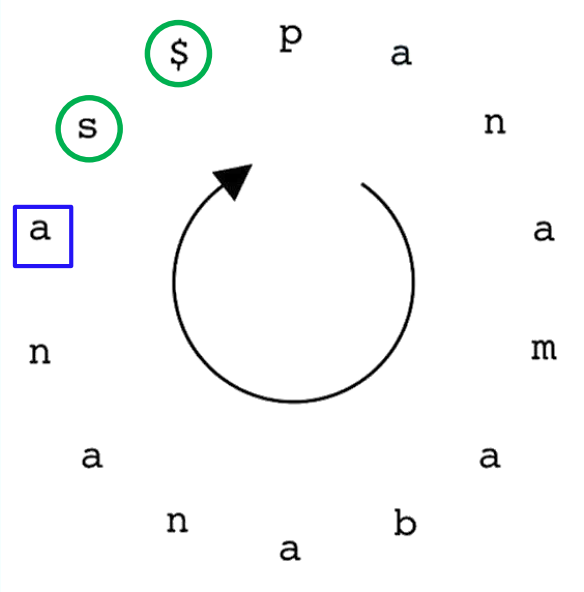
More Efficient BWT Decompression



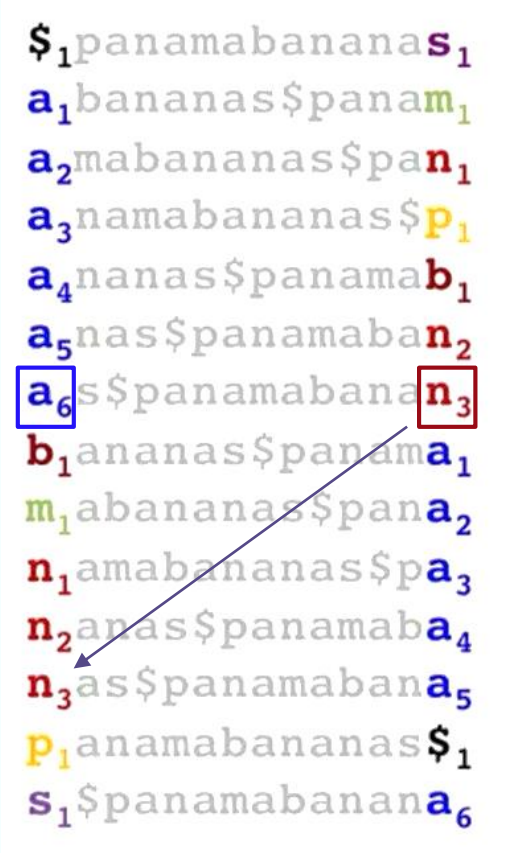


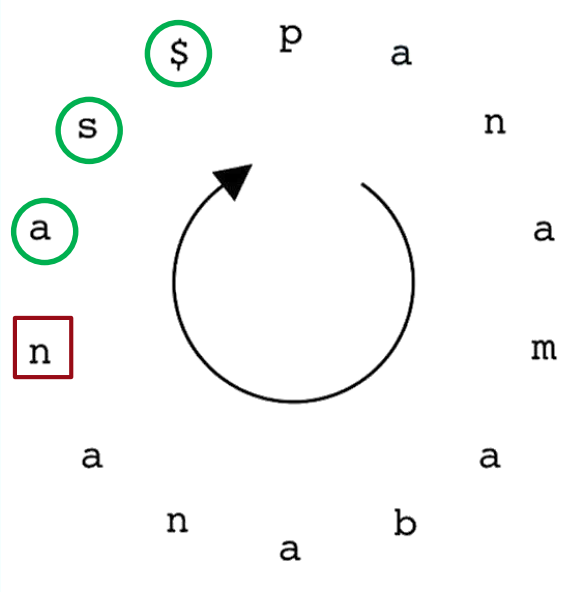
More Efficient BWT Decompression



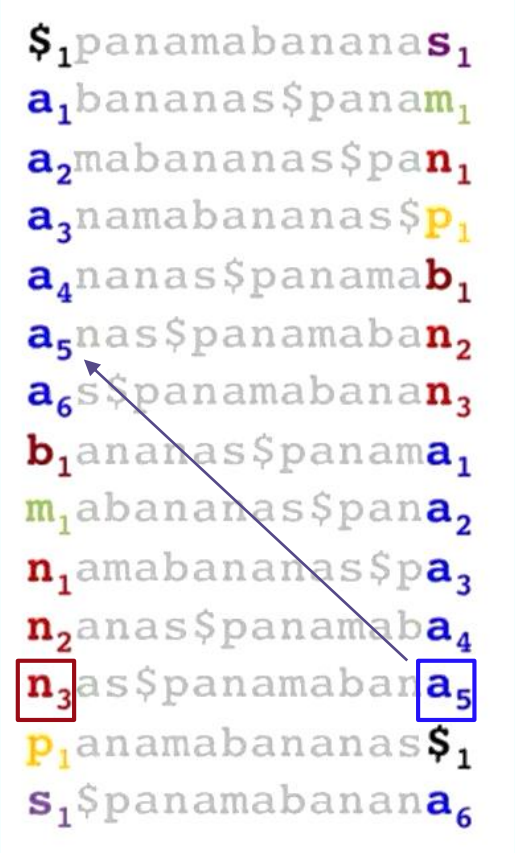


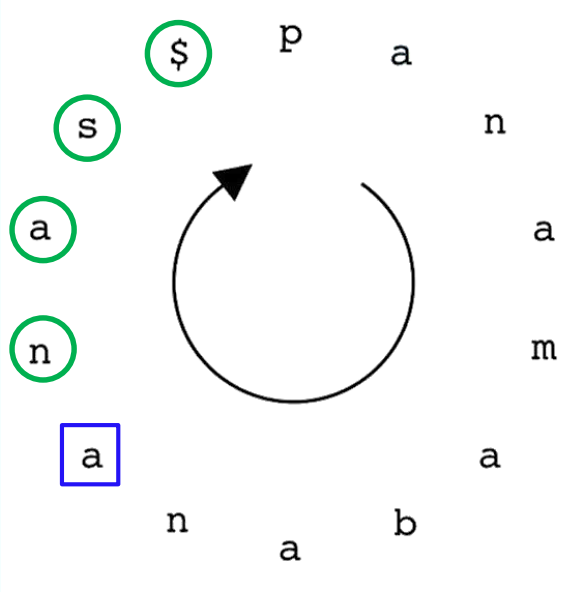
More Efficient BWT Decompression



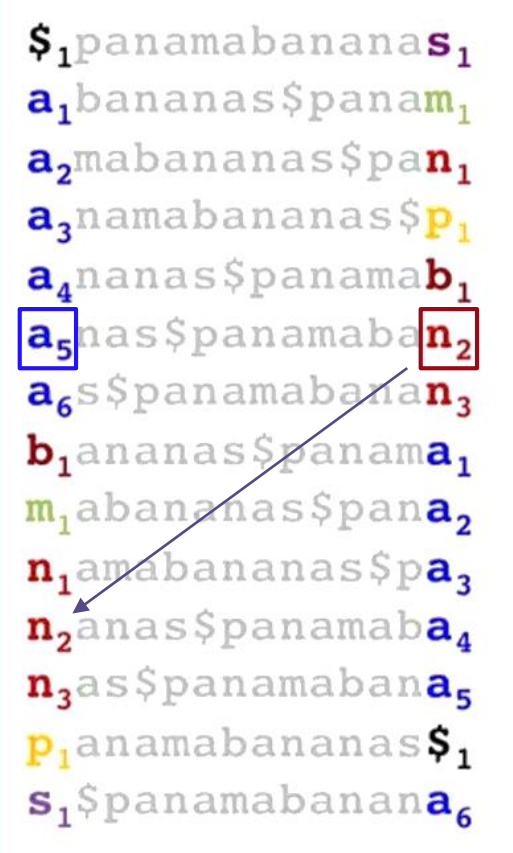


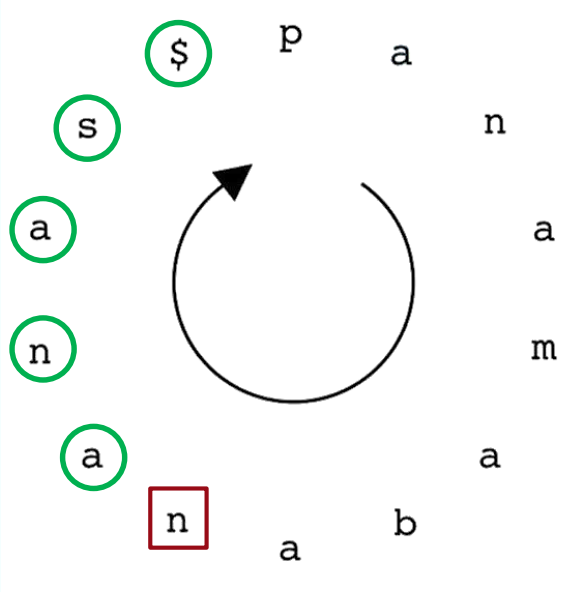
More Efficient BWT Decompression





More Efficient BWT Decompression

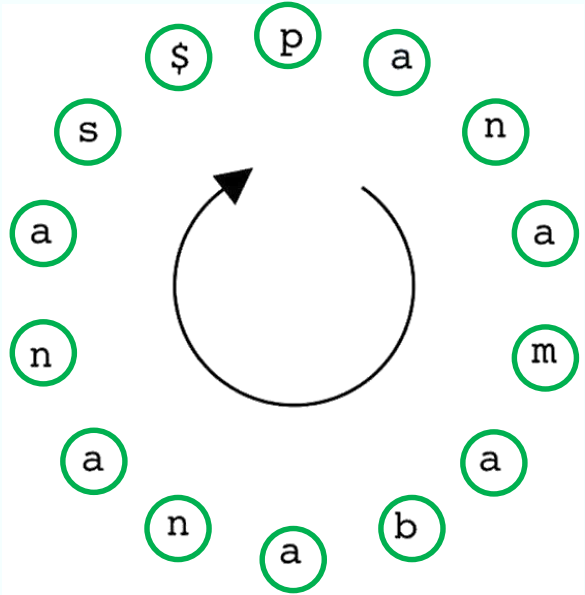




And So On …

More Efficient BWT Decompression

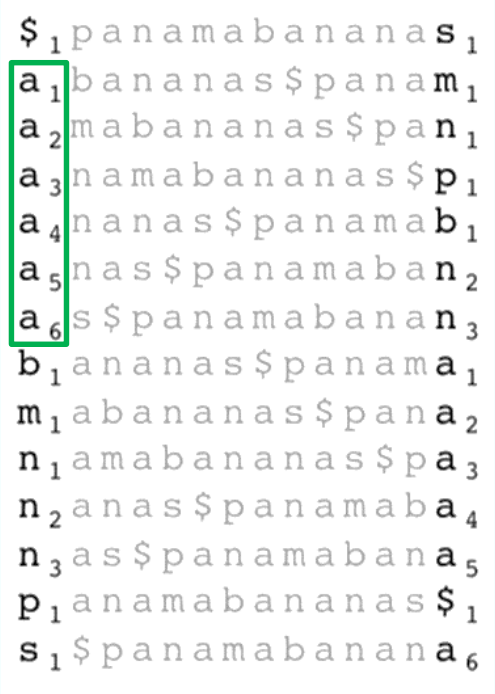
$_1$panamabananas$_1$
a$_1$bananas$panam$_1$
a$_2$mabananaspan_1$
a$_3$namabananas$p$_1$
a$_4$nanas$panamab$_1$
a$_5$nas$panamaban$_2$
a$_6$s$panamabanan$_3$
b$_1$ananas$panama$_1$
m$_1$abananas$pana$_2$
n$_1$amabananaspa_3$
n$_2$anas$panamab$_4$
n$_3$as$panamaban$_5$
p$_1$anamabananas$$_1$
s$_1$$panamabanan$_6$



Memory: 2|Genome|=O(|Genome|)

Finding Pattern Matches Using BWT

Searching for ana in "panamabananas$"

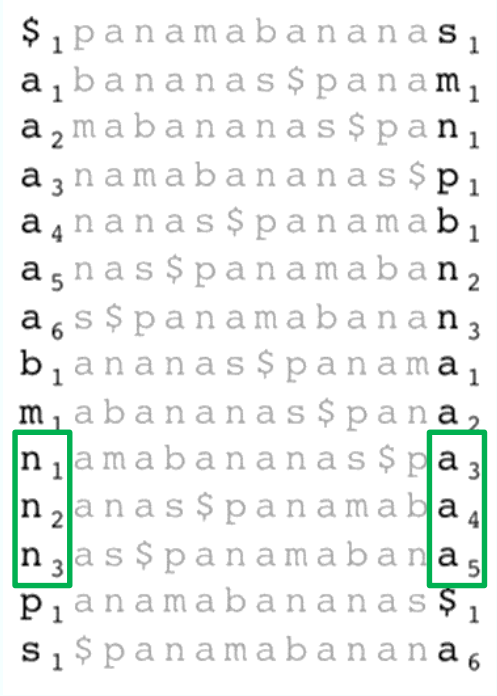


Finding Pattern Matches Using BWT

Searching for ana in "panamabanana$"

Finding Pattern Matches Using BWT

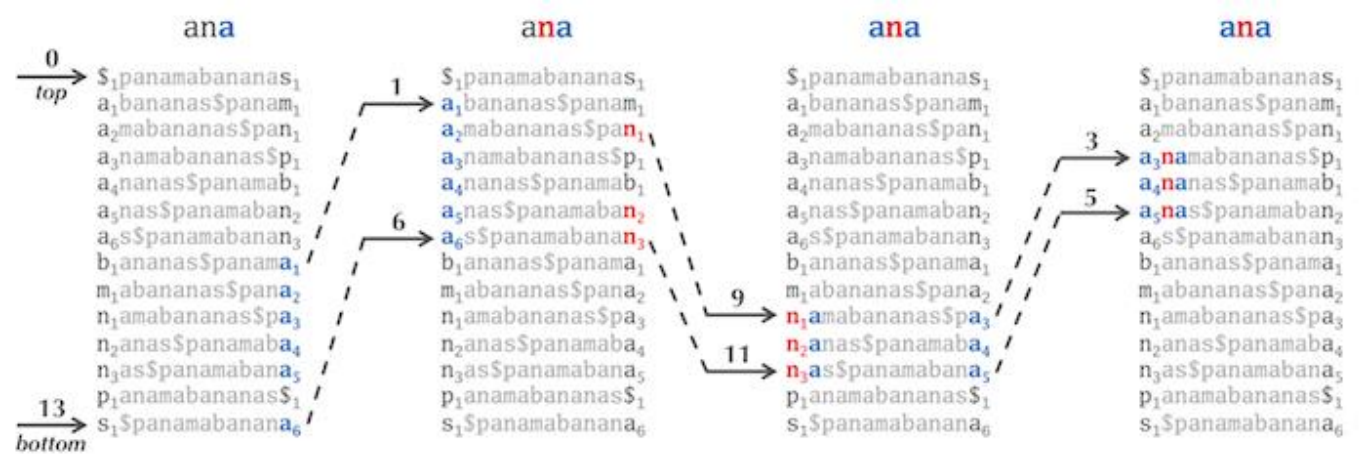
Searching for **ana** in "panamabanana$"



Finding Pattern Matches Using BWT

Searching for **ana** in "panamabanana$"

The Last to First Mapping



i	*FirstColumn*	*LastColumn*	LastToFirst(*i*)
0	$1	s1	13
1	a1	m1	8
2	a2	n1	9
3	a3	p1	12
4	a4	b1	7
5	a5	n2	10
6	a6	n3	11
7	b1	a1	1
8	m1	a2	2
9	n1	a3	3
10	n2	a4	4
11	n3	a5	5
12	p1	$1	0
13	s1	a6	6

BWMatching

```
BWMatching(LastColumn, Pattern, LastToFirst)
    top ← 0
    bottom ← |LastColumn| - 1
    while top ≤ bottom
        if Pattern is nonempty
            symbol ← last letter in Pattern
            remove last letter from Pattern
            if positions from top to bottom in LastColumn contain an occurrence of symbol
                topIndex ← first position of symbol among positions from top to bottom in LastColumn
                bottomIndex ← last position of symbol among positions from top to bottom in LastColumn
                top ← LastToFirst(topIndex)
                bottom ← LastToFirst(bottomIndex)
            else
                return 0
        else
            return bottom - top + 1
```

We can Speed up the last to first mapping part by substituting it with count arrays.

Where Are the Matches?

We know that ana occurs 3 times, but where?

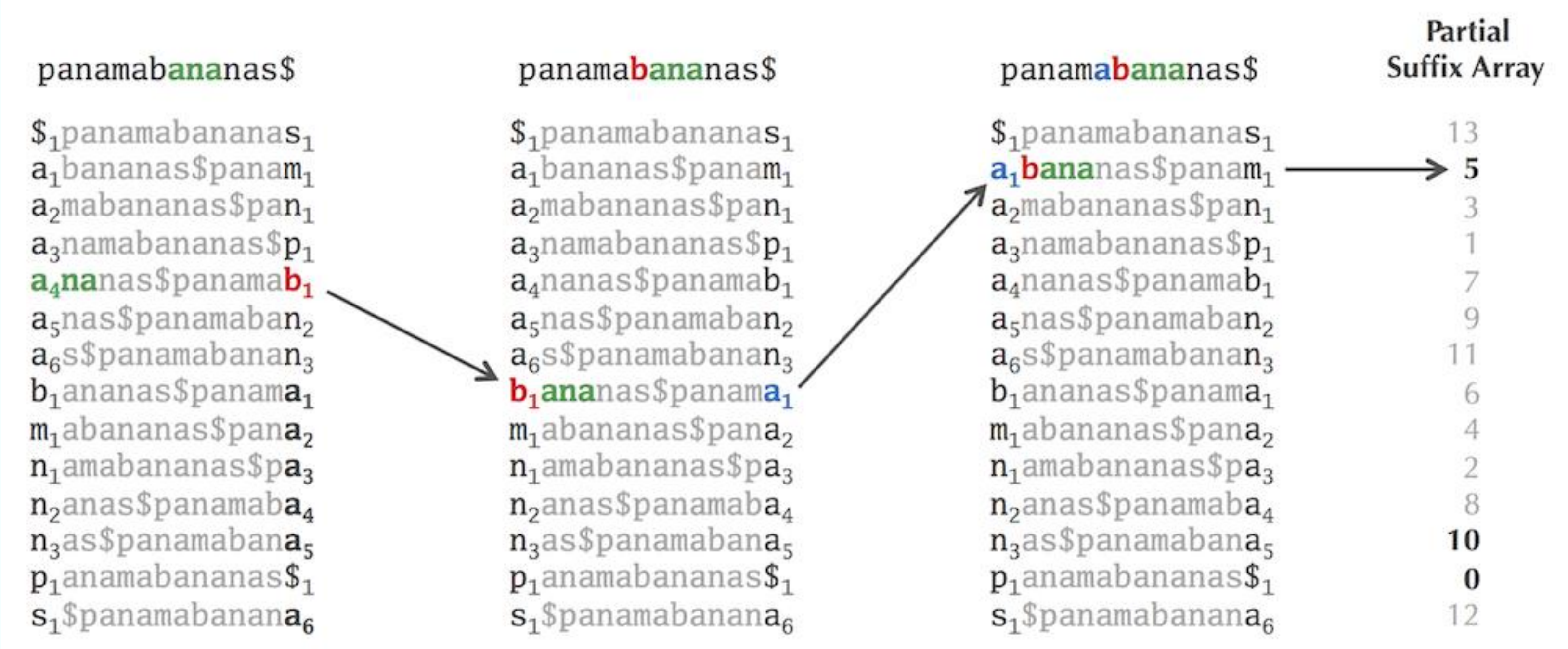
Suffix Arrays:
holds starting position of each suffix beginning a row.
Memory About 4 * |Genome|

Thus, ana occurs at positions 1,7,9 of
"panamabanana$"

Memory once again

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

partial suffix array



Inexact Matching

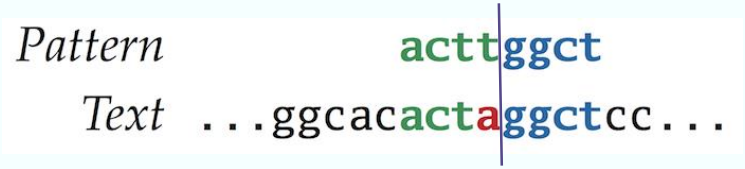
- Finding exact matches was not our original goal!
- We need to look at INEXACT matching in order to find variants(**SNPs**)

Multiple Approximate Pattern Matching Problem: *Find all approximate occurrences of a collection of patterns in a text.*

•**Input:** A string *Text*, a collection *Patterns* of (shorter) strings, and an integer *d*.
•**Output:** All starting positions in *Text* where a string from *Patterns* appears as a substring with at most *d* mismatches.

Method 1: Seeding

- If Pattern occurs in Text with **a single mismatch**, then we can divide Pattern into two halves, **one of which occurs exactly in Text**.



- If we find a match with one of these halves of Pattern, we then check if the entire string Pattern occurs with a single mutation.

Theorem:
If pattern occurs in Genome with **d mismatches**, then we can divide pattern into **d+1 "equal" pieces** and find **at least one exact match.**

Method 1: Seeding

- Thus for example, if we are looking for a **pattern of length 20** with at most **d = 3 mismatches**, then we can divide this pattern into **four parts of length 20/(3+1)=5** and search for exact matches of these shorter substrings:

- This observation is helpful because it reduces approximate pattern matching to the exact matching of shorter patterns, which allows us to use fast algorithms that are designed for exact pattern matching but are not applicable to approximate pattern matching, such as approaches based on suffix trees and suffix arrays.

Method 1: Seeding

- Say we are looking for **at most d mismatches**.
- Divide each of our patterns 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.

Method 2: BWT

Recall: Searching for ana ← in "panamabananas$"

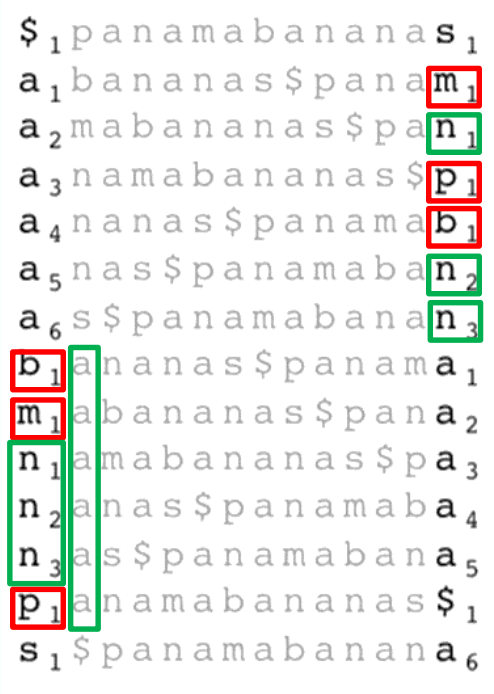
If we allow one mismatch, then we need to keep the red letters around.

mismatches
1
0
1
1
0
0

Method 2: BWT

Recall: Searching for ana in "panamabananas$"

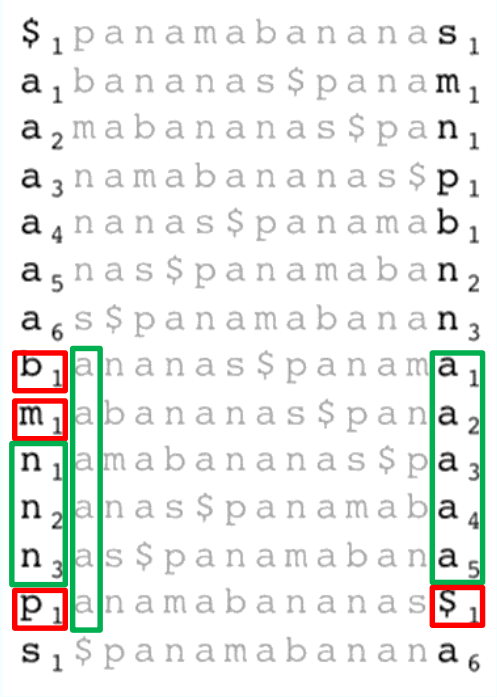
Now we extend all strings with at most 1 mismatch



Method 2: BWT

Recall: Searching for ana ← in "panamabananas$"

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



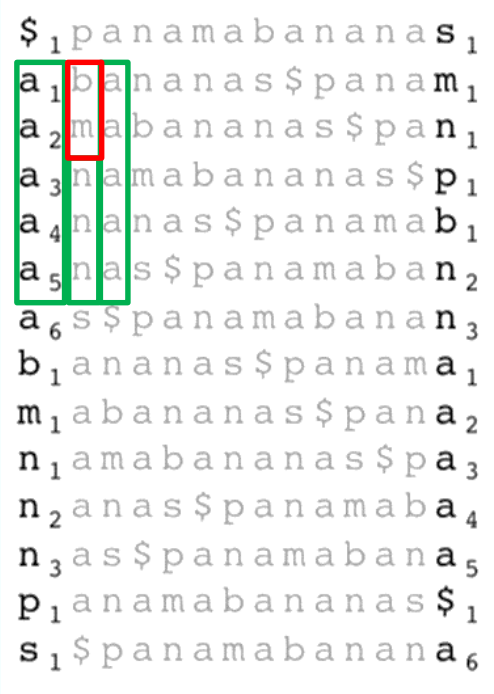
mismatches

1
1
0
0
0
2

Method 2: BWT

Recall: Searching for ana ← in "panamabananas$"

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



mismatches

1
1
0
0
0

RESOURCES

- Compeau, Phillip, and Pave Pevzner. "Bioinformatics algorithms: an active learning approach, Chapter 9: How do we locate disease causing mutation?