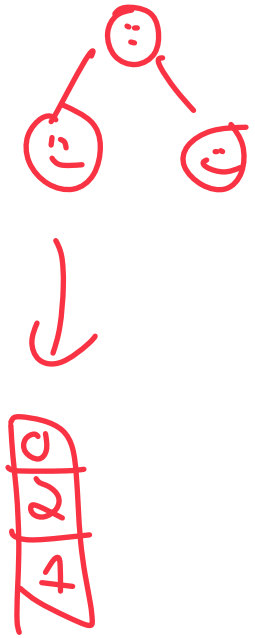
String Algorithms and Data Structures

Suffix Arrays

CS 199-225
Brad Solomon

October 21, 2024

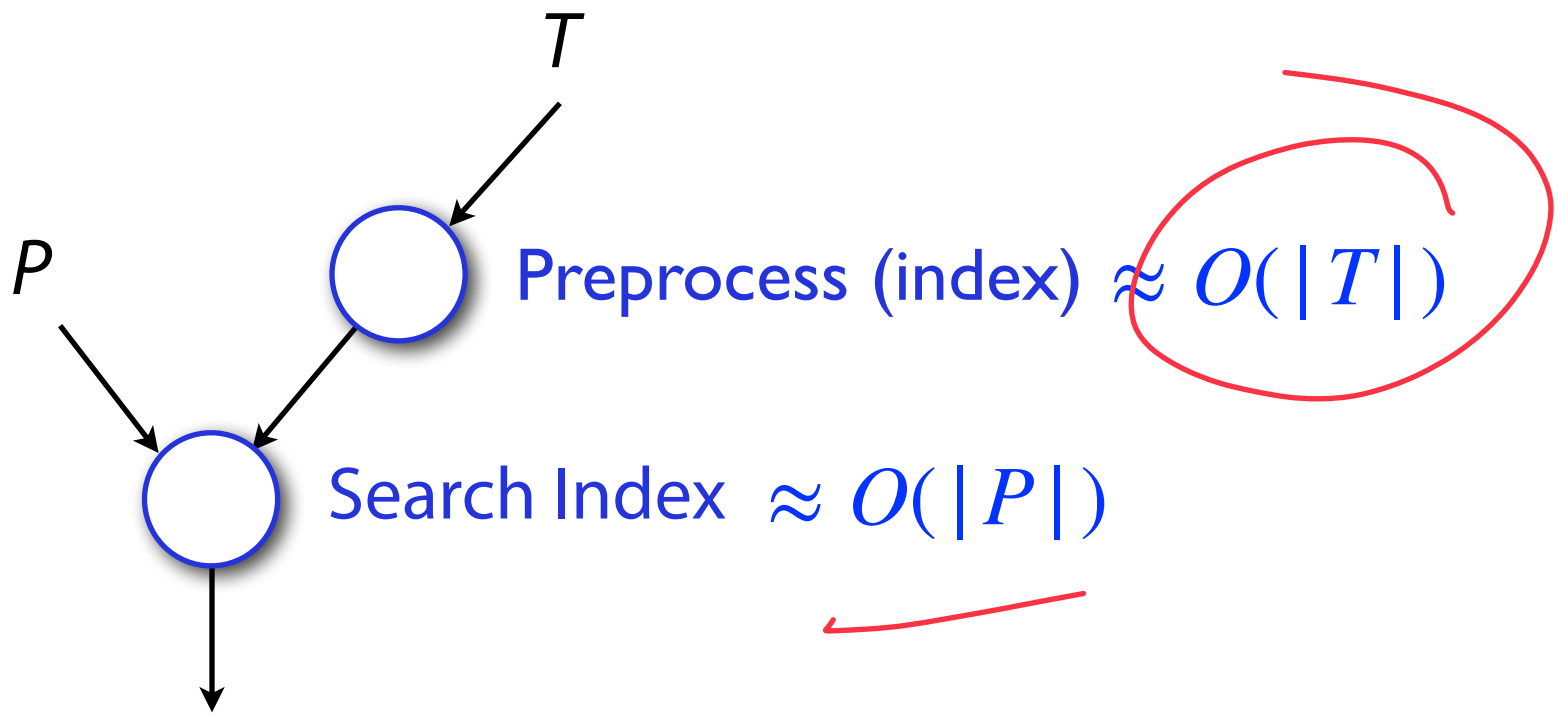


UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Exact pattern matching w/ *indexing*

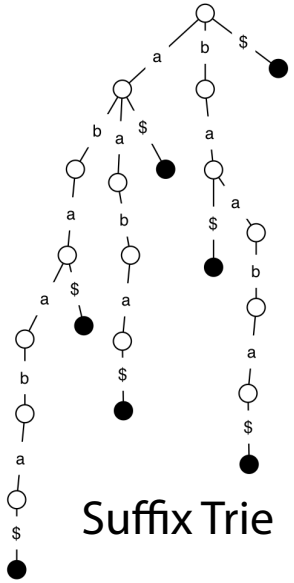
$|T| >> |P|$



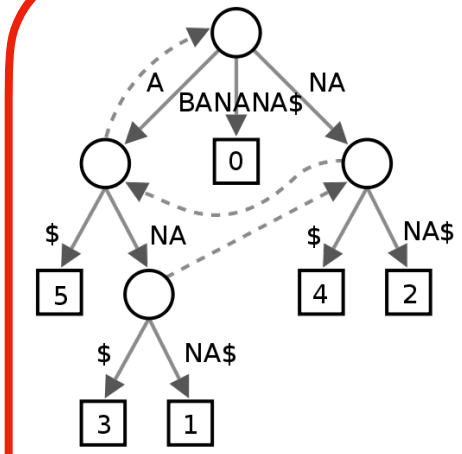
Exact pattern matching *w/ indexing*

There are many data structures built on ***suffixes***

Modern methods still use these today



Suffix Trie



Suffix Tree

6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

Suffix Array

$ 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

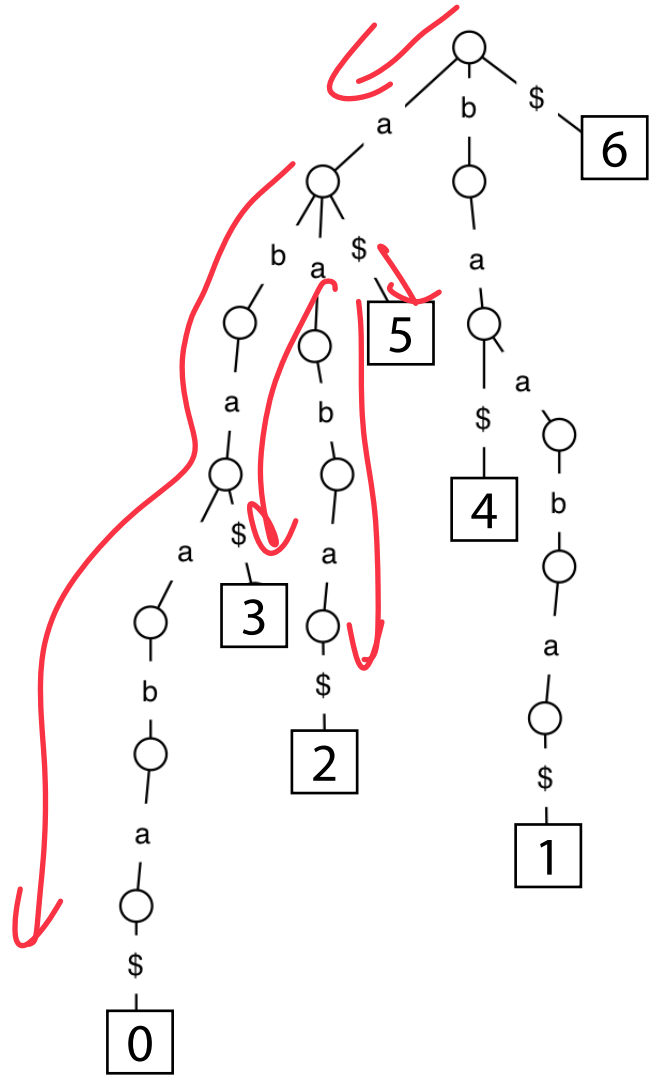
FM Index

Suffix Trie

A rooted tree storing a collection of suffixes as (key, value) pairs

The tree is structured such that:

- Each key is "spelled out" along some path starting at root
- Each edge is labeled with a character $c \in \Sigma$
- For given node, at most one child edge has label c, for any $c \in \Sigma$
- Each key's value is stored at a leaf



Suffix Tree

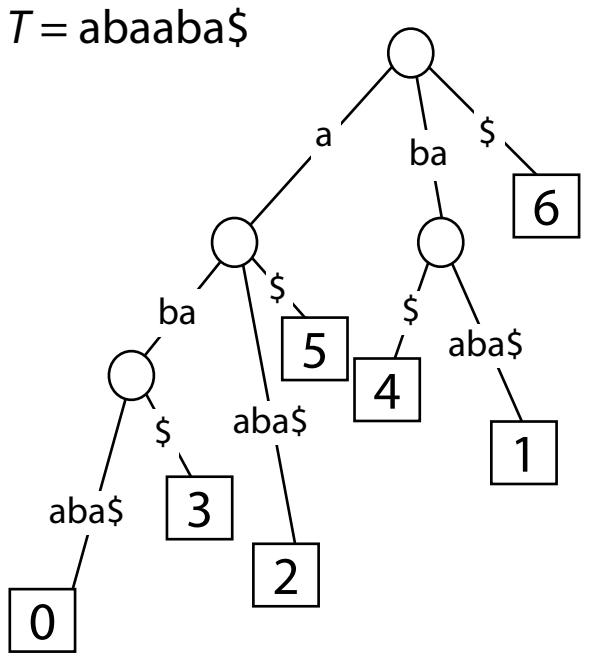
A rooted tree storing a collection of suffixes as (key, value) pairs

The tree has many similarities to the trie but:

Each edge is labeled with *a string s*

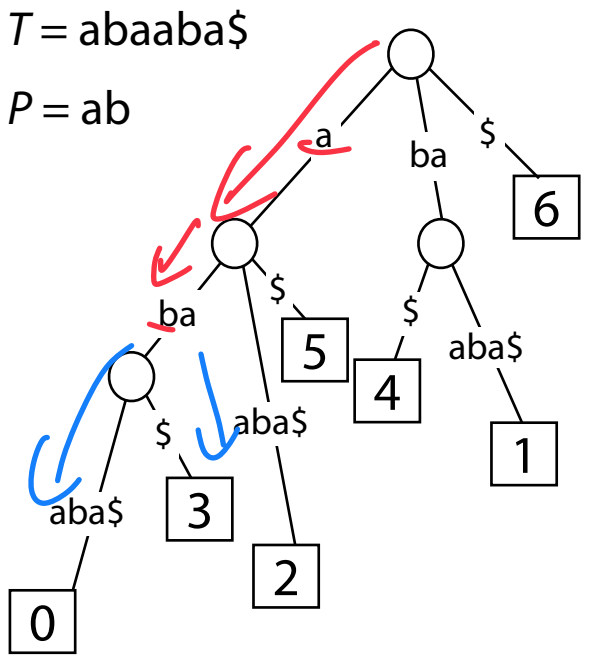
For given node, at most one child edge *starts with character c*, for any $c \in \Sigma$

Each internal node contains >1 children



Searching a suffix tree

How efficient is search?



Claim: To find **k** leaves, we have to traverse **<= k-1** internal nodes

Searching a suffix tree

How efficient is search?

Claim: To find **k** leaves, we have to traverse **<= k-1** internal nodes

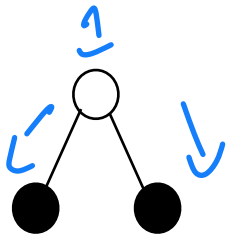
Base Case:

Searching a suffix tree

How efficient is search?

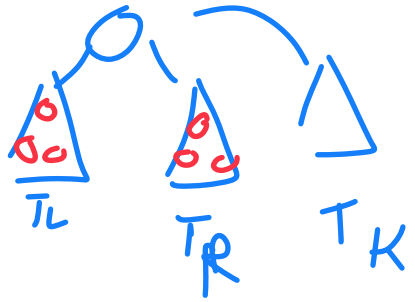
Claim: To find **k** leaves, we have to traverse **<= k-1** internal nodes

Base Case: 1 internal node



Find two leaves, traverse 1 node!

Searching a suffix tree



How efficient is search?

Claim: To find **k** leaves, we have to traverse **<= k-1** internal nodes

Induction: Assume any tree w/ **m < N** leaves has at most **m-1** internal nodes

Split the **N leaf** tree into two subtrees with **i** and **N-i** leaves respectively

These subtrees will have **i-1 and N-i-1** internal nodes (and the root is 1)

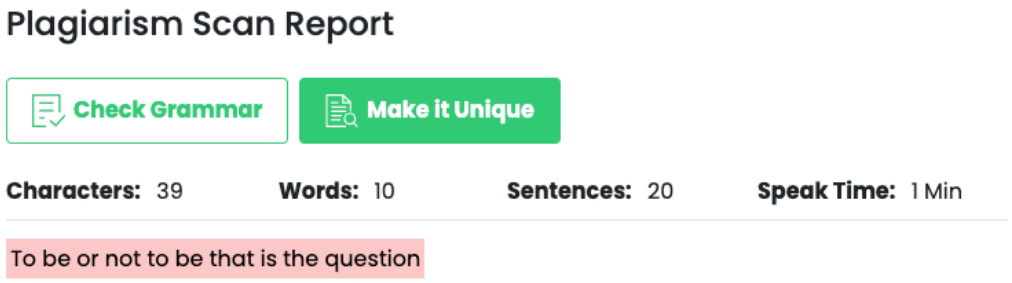
Number internal nodes $= (i-1) + (N-i-1) + 1 = N-1$

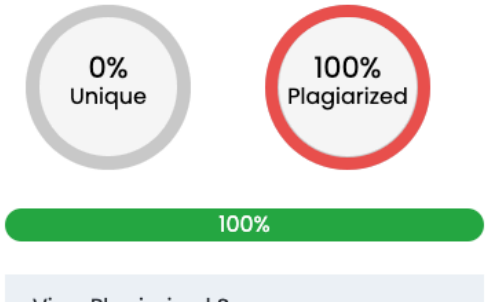
Suffix trie vs suffix tree: bounds

	Suffix trie	**Suffix tree**
Time: Does P occur?	$O(n)$	$O(n)$
Time: Report k locations of P	$O(n + m^2)$	$O(n + k)$
Space	$O(m^2)$	$O(m)$

$m = |T|$, $n = |P|$, $k =$ # occurrences of P in T

Suffix trees in the real world

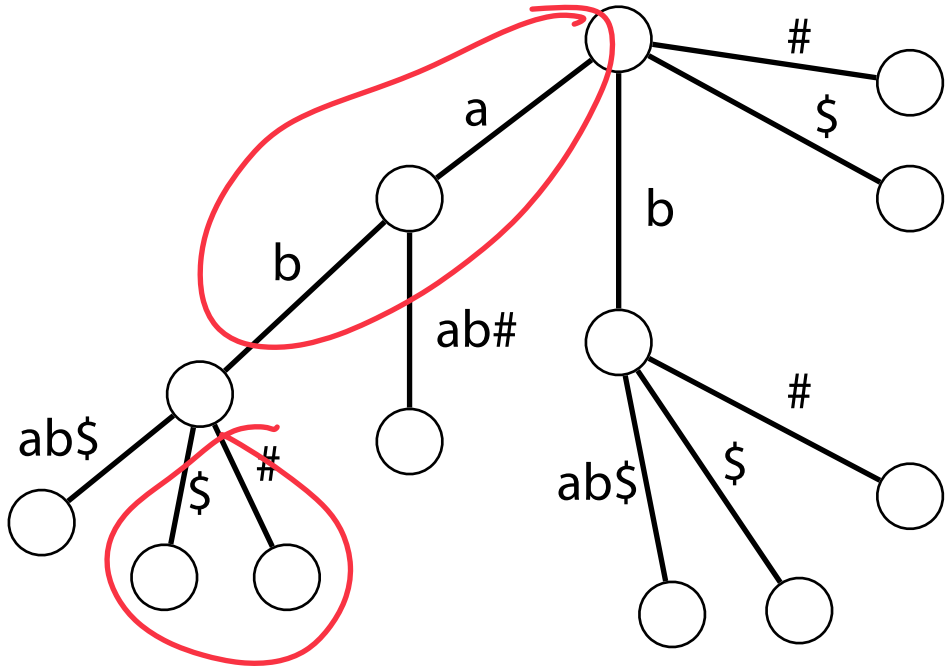




Two Break characters

X = ABAB Y = AAB

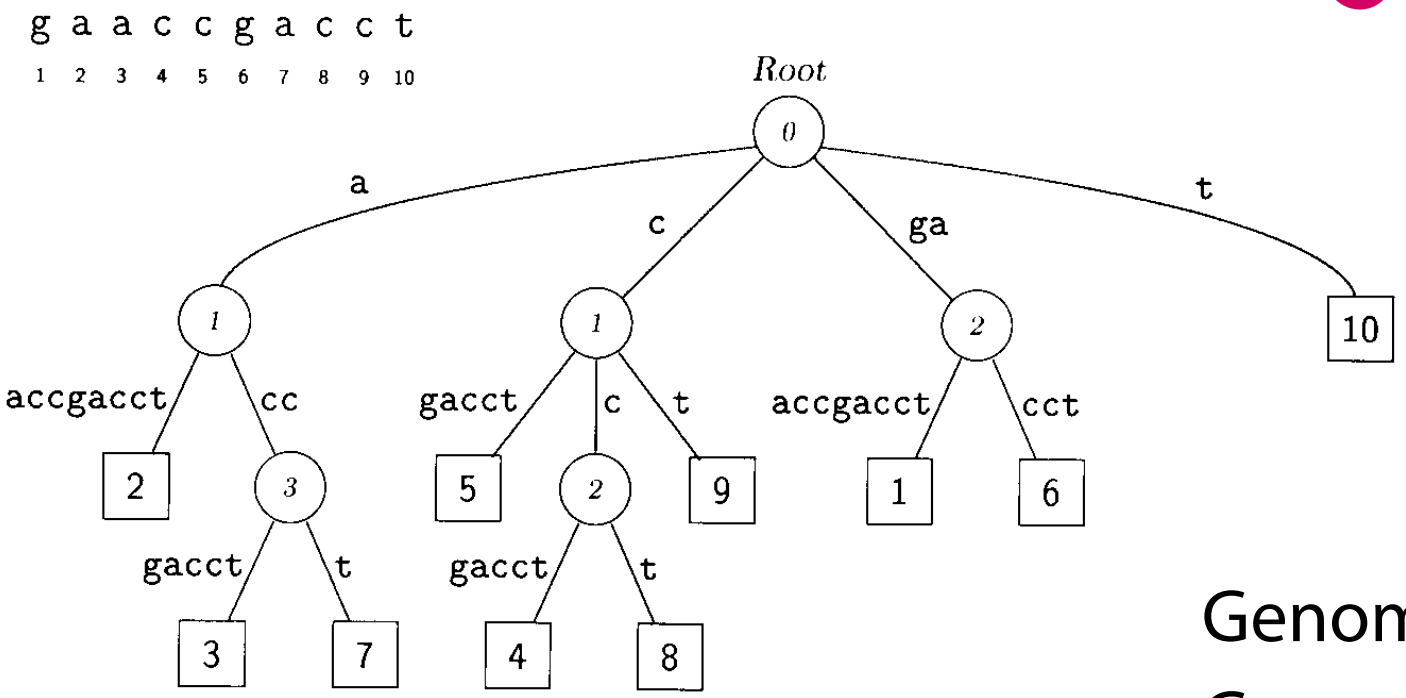
Generalized Suffix Tree: ABAB$AAB#



Suffix trees in the real world

ancestry®

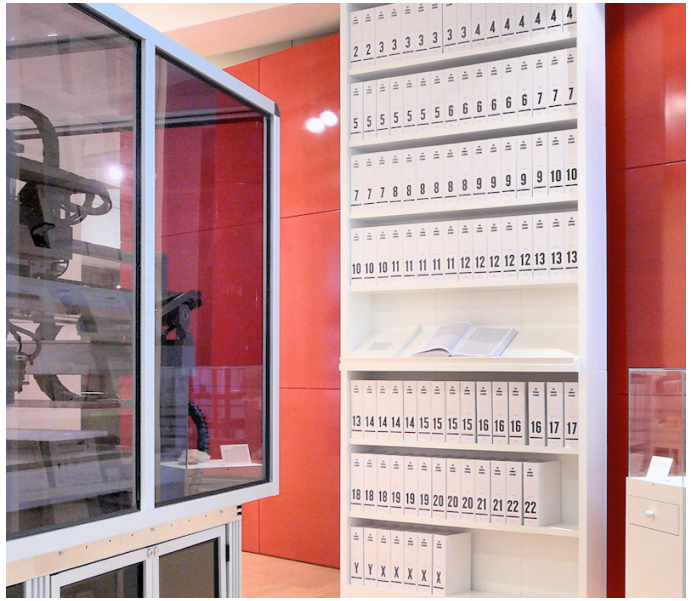
23andMe

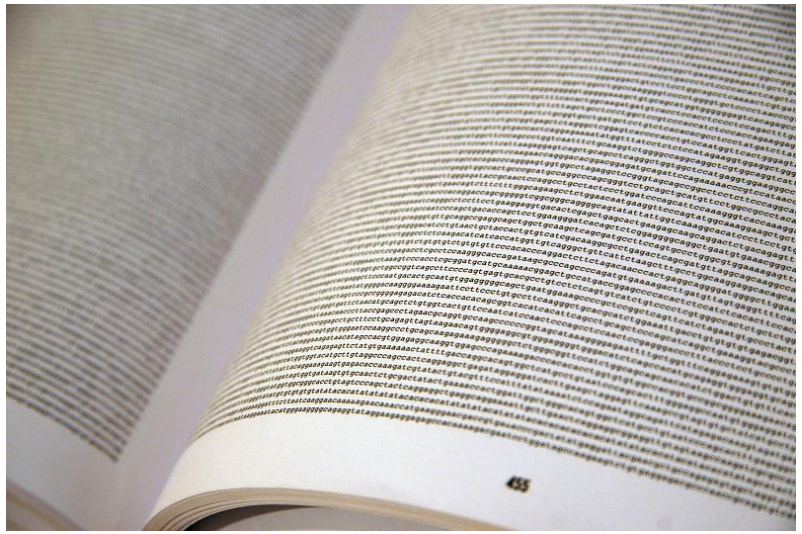


Genome A: TCGATGCGAGGATCATTA

Genome B: AAGTCGCGAGGATCACCG

Suffix trees in the real world: MUMmer





Delcher, Arthur L., et al. "Alignment of whole genomes." *Nucleic Acids Research* 27.11 (1999): 2369-2376.

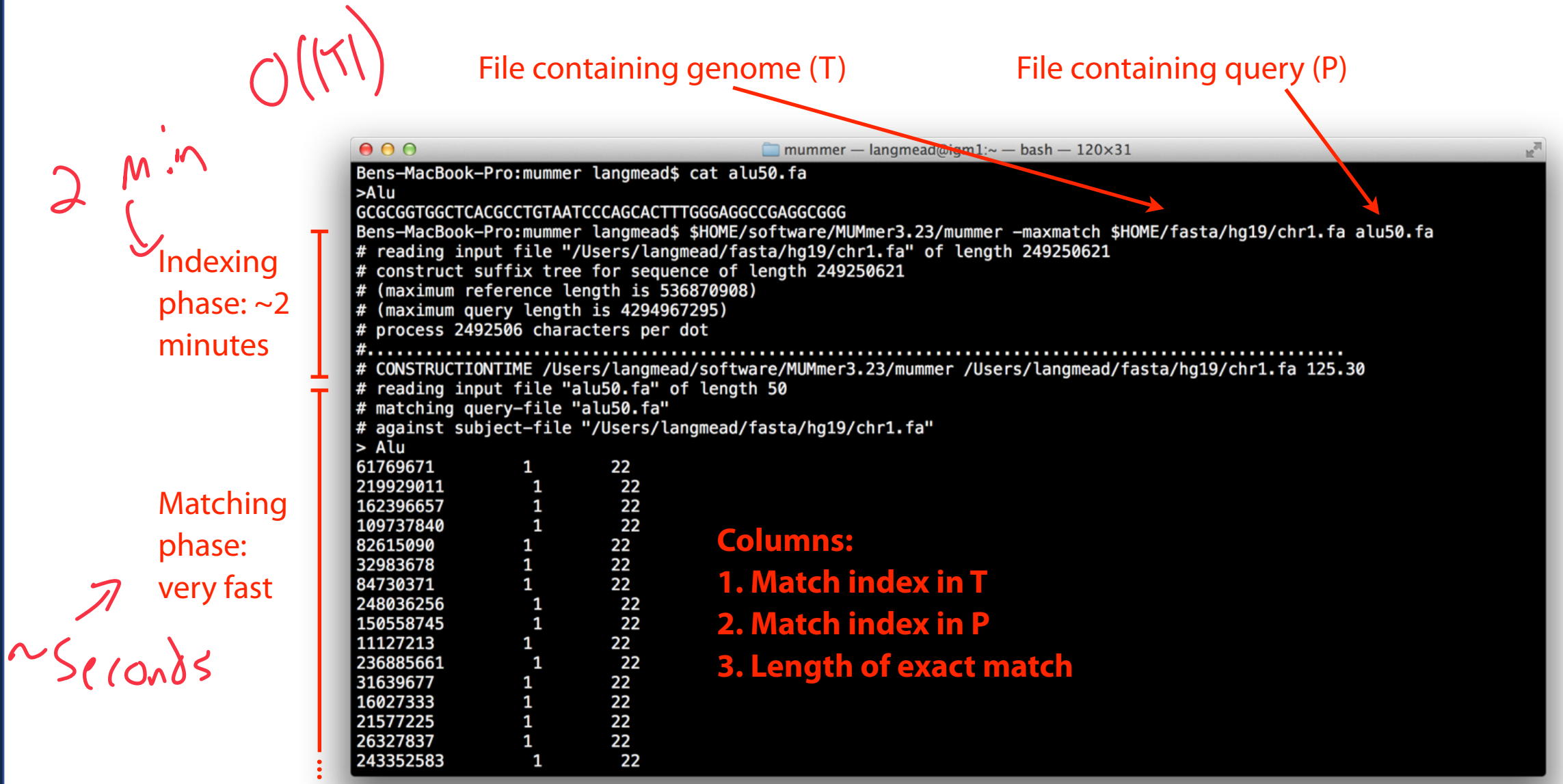
Delcher, Arthur L., et al. "Fast algorithms for large-scale genome alignment and comparison." *Nucleic Acids Research* 30.11 (2002): 2478-2483.

Kurtz, Stefan, et al. "Versatile and open software for comparing large genomes." *Genome Biol* 5.2 (2004): R12.

~ 4,000 citations

http://mummer.sourceforge.net

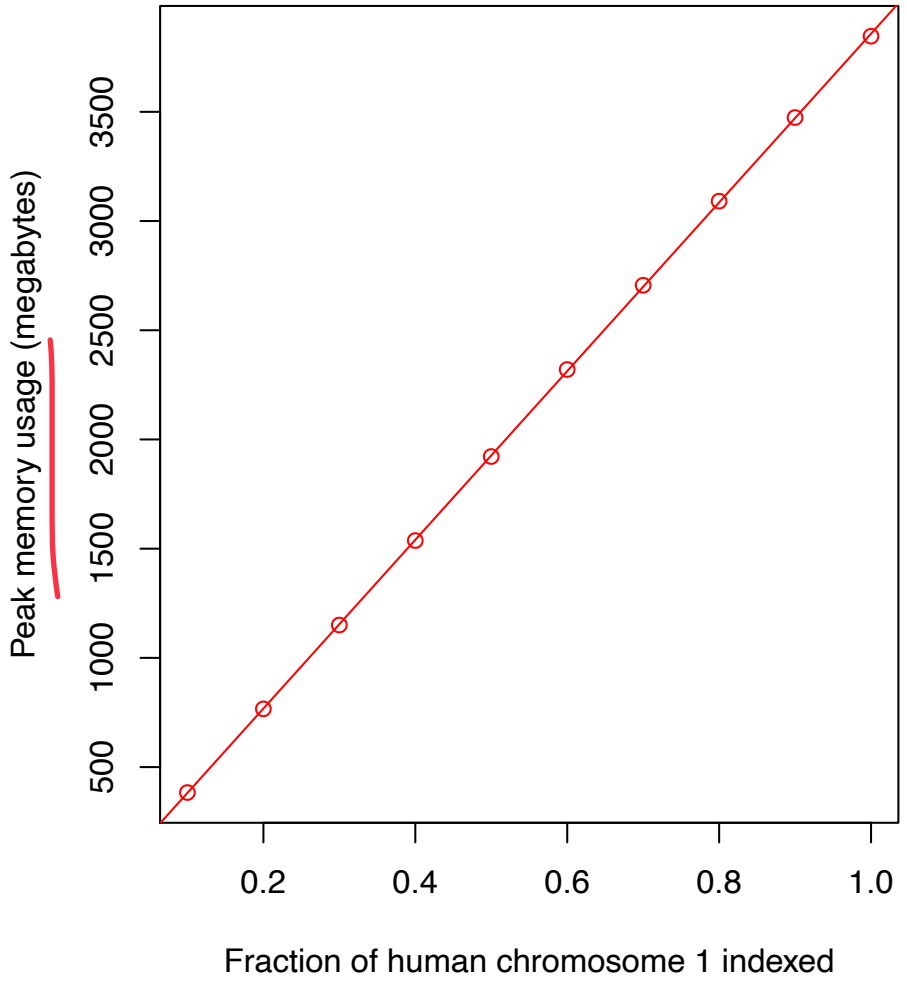
Suffix trees in the real world: MUMmer

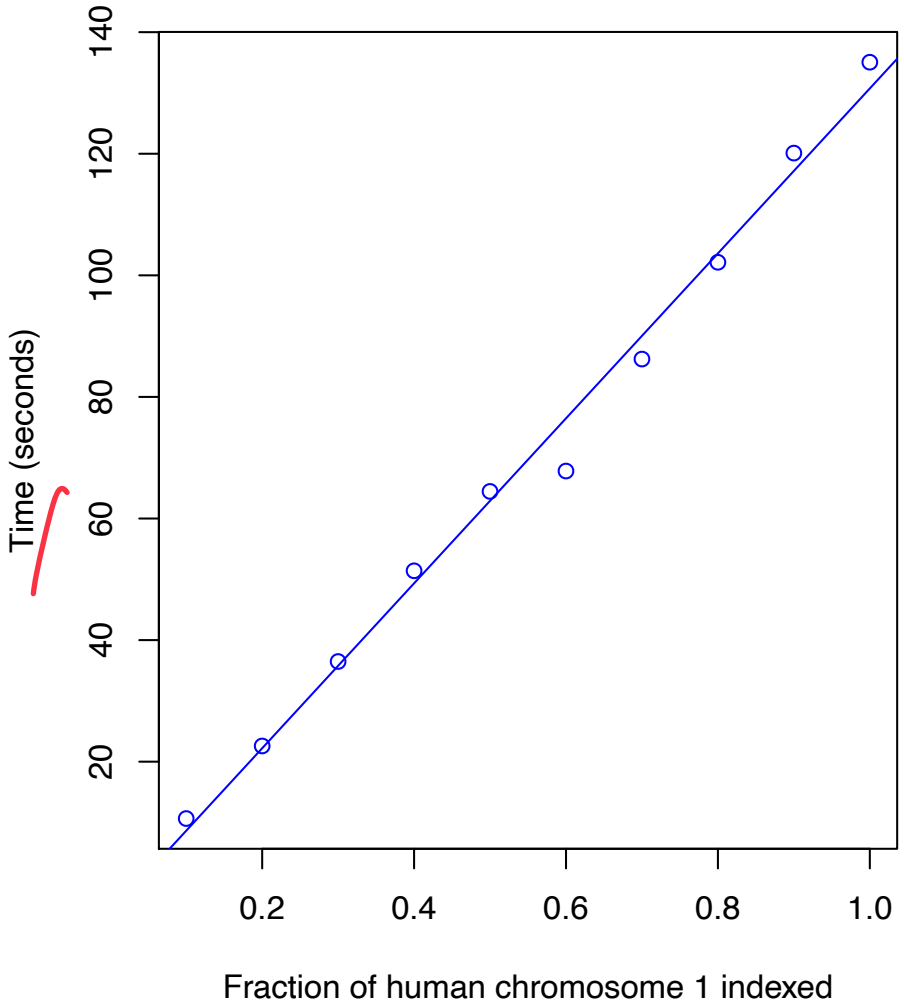


Example by Ben Langmead

Suffix trees in the real world: MUMmer

Linear in space & time!



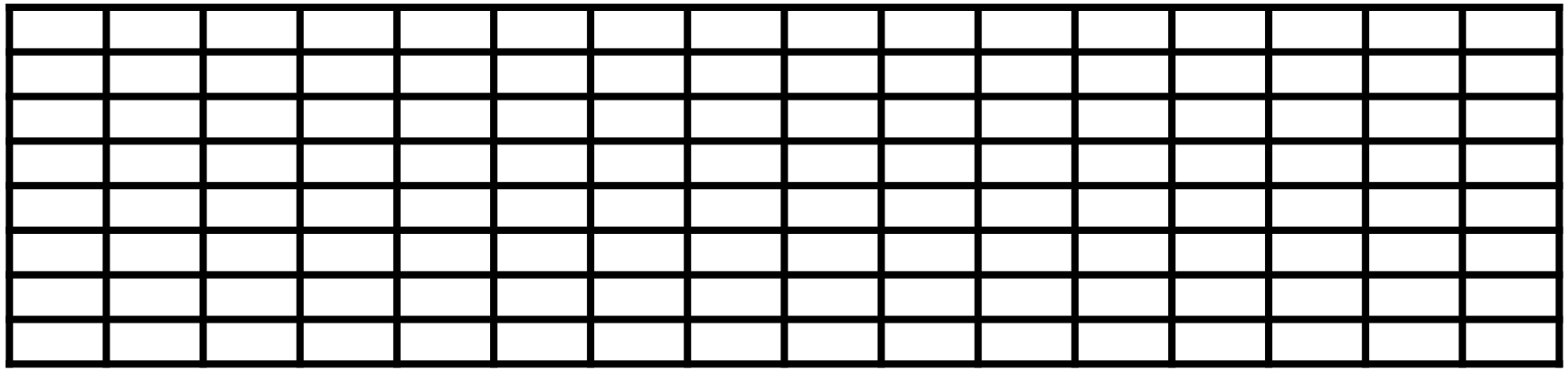


For whole chromosome 1, took 2m:14s and used 3.94 GB memory

Suffix trees in the real world: constant factor

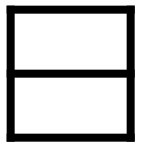
Suffix Trees are O(|T|) but there's a hidden constant factor at work:

MUMmer constant factor ≈ **15.76 bytes per nt**



Suffix tree of human genome: **>45 GB**

'Raw' two-bit encoding ≈ **2 bits per nt**



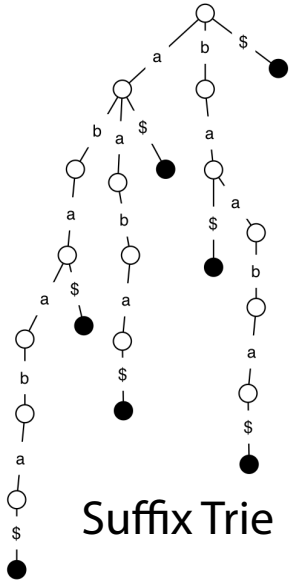
Raw encoding of human genome: **~0.75 GB**

A C G T

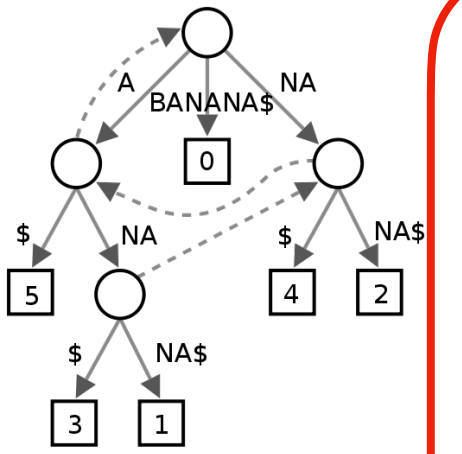
Exact pattern matching *w/ indexing*

There are many data structures built on ***suffixes***

More efficient to store, less efficient* to use



Suffix Trie



Suffix Tree

6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

Suffix Array

$ 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

FM Index

Lexicographic Order

A systematic way of organizing strings by the content and arrangement of its characters

Lexicographic Order

A systematic way of organizing strings by the **content** and arrangement of its characters

Strings are compared by their individual characters.

Alphabetical Order `A < B < ... < Z`

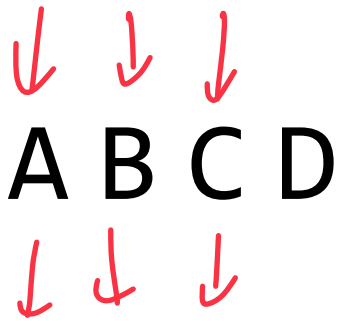
ASCII Order `$ < 0 < A < a`

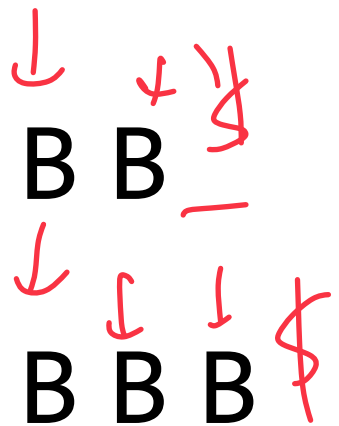
ASCII Value	Character
36	$
…	…
48	0
…	…
65	A
…	…
97	a

Lexicographic Order

A systematic way of organizing strings by the content and **arrangement** of its characters

Characters are compared in order from left to right





ABAB < ABCD < BB < BBB

Lexicographic Order

A systematic way of organizing strings by the **content** and **arrangement** of its characters

What is the *lexicographically* smallest string?

A) “beep”

B) “zzz”

C) “aardvarks”

D) “apples”

Lexicographic Order

A systematic way of organizing strings by the **content** and **arrangement** of its characters

$ the smallest char possible

What is the *lexicographically* smallest string?

A) “bah$”

B) “x”

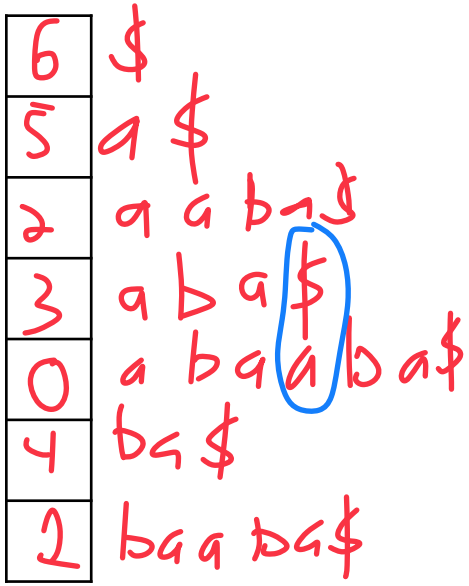
C) “bb$”

D) “b$b”

Suffix Array

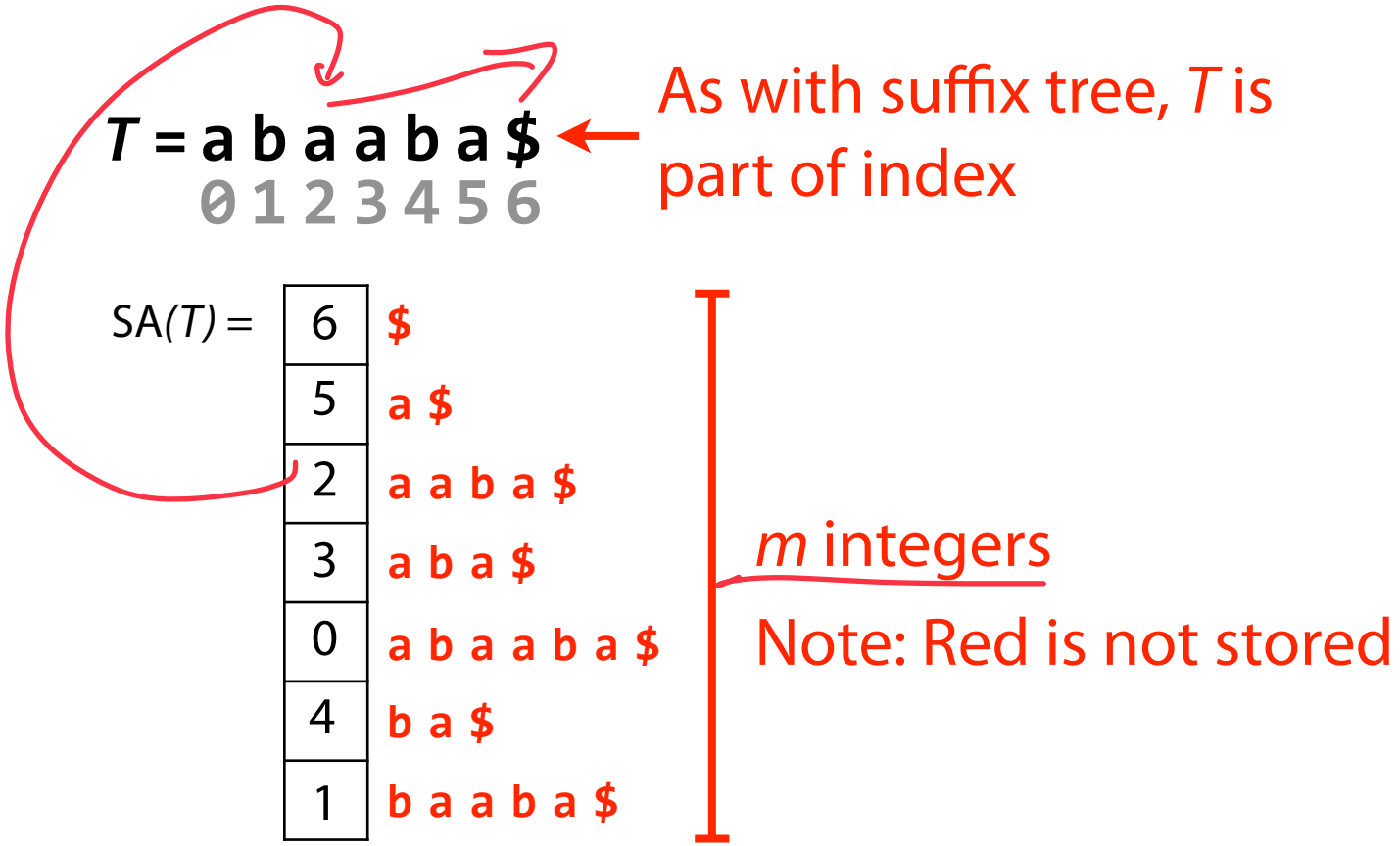
Suffix array of *T* is an array of integers specifying lexicographic (alphabetical) order of *T*'s suffixes

T = a b a a b a $
0 1 2 3 4 5 6



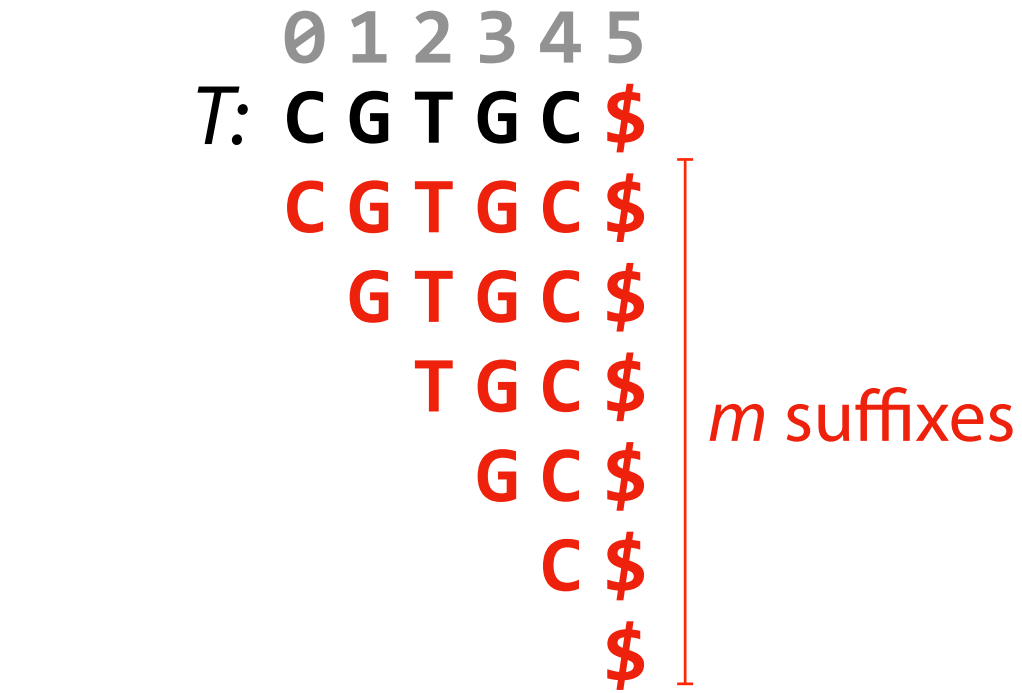
Suffix Array

Suffix array of *T* is an array of integers specifying lexicographic (alphabetical) order of *T*'s suffixes

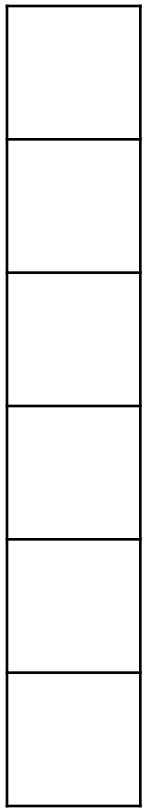


vector<int> build_sarray(string T)

Input:



Output:



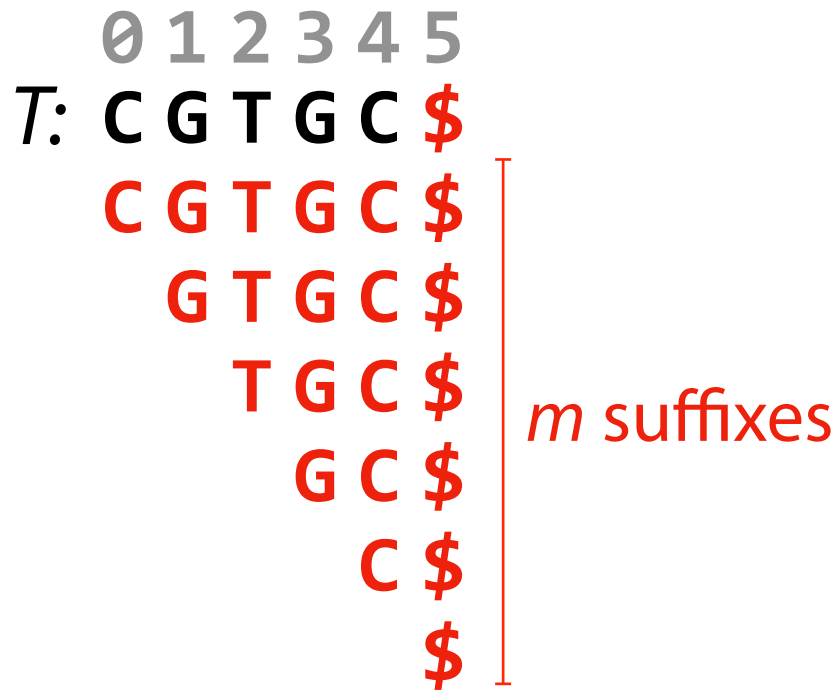
Take string

1) get all suffixes

2) sort suffixes
   - store start index

vector<int> build_sarray(string T)

Input:



Output:

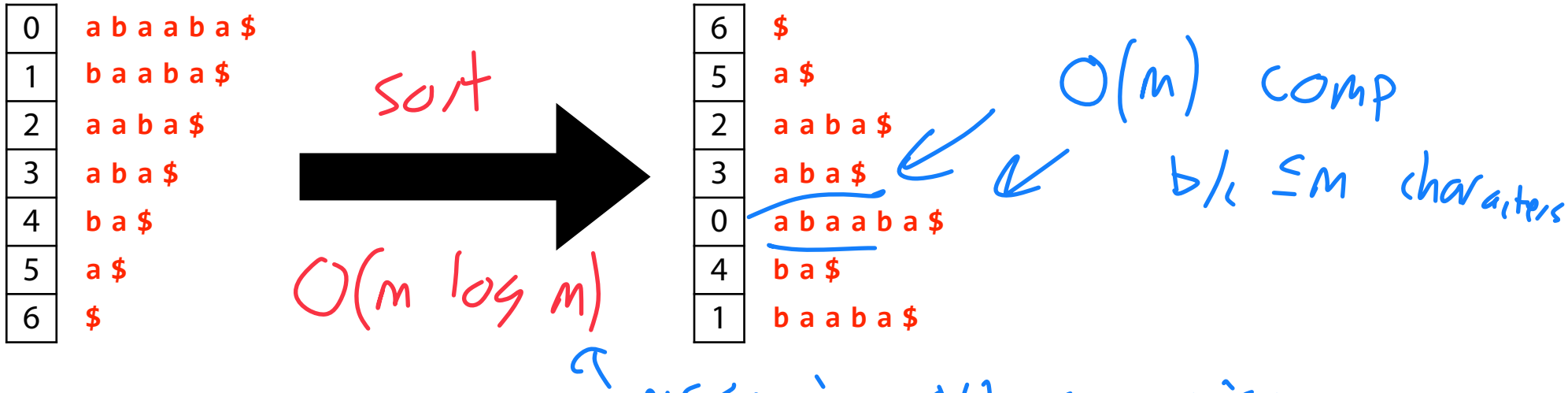
5
4
0
3
1
2

Suffix array: build by sorting (from array)

$T = m$

$P = n$

Use your favorite sort, e.g., quickSort, heapSort, insertSort, …



Expected time: $O(m^2 \log m)$

A A A$
A A $

Suffix array: build by sorting ***suffixes***

Another idea: Use a sort algorithm that's aware that the items being sorted are all suffixes of the same string

Original suffix array paper suggested an ***O*(*m* log *m*)** algorithm

Manber U, Myers G. "Suffix arrays: a new method for on-line string searches." SIAM Journal on Computing 22.5 (1993): 935-948.

Other popular ***O*(*m* log *m*)** algorithms have been suggested

Larsson NJ, Sadakane K. Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1-43/(1999), Department of Computer Science, Lund University, Sweden, 1999.

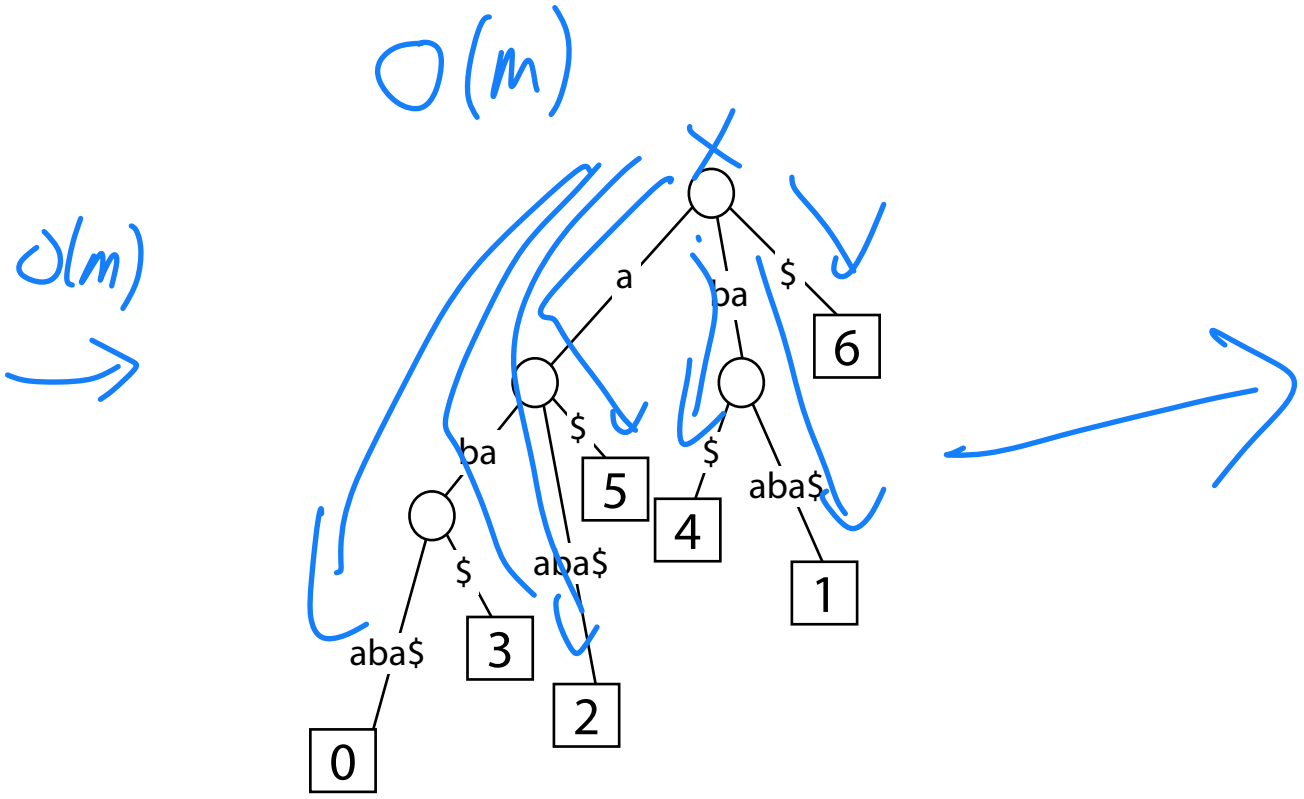
There exist several ***O*(*m*)** algorithms that *divide-and-conquer*

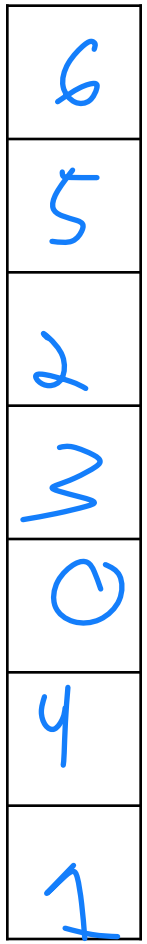
Kärkkäinen J, Sanders P. "Simple linear work suffix array construction." Automata, Languages and Programming (2003): 187-187.

Ko P, Aluru S. "Space efficient linear time construction of suffix arrays." *Combinatorial Pattern Matching*. Springer Berlin Heidelberg, 2003.

Suffix array: build by suffix tree

(a) Build suffix tree, (b) traverse in lexicographic order, (c) upon reaching leaf, append suffix to array



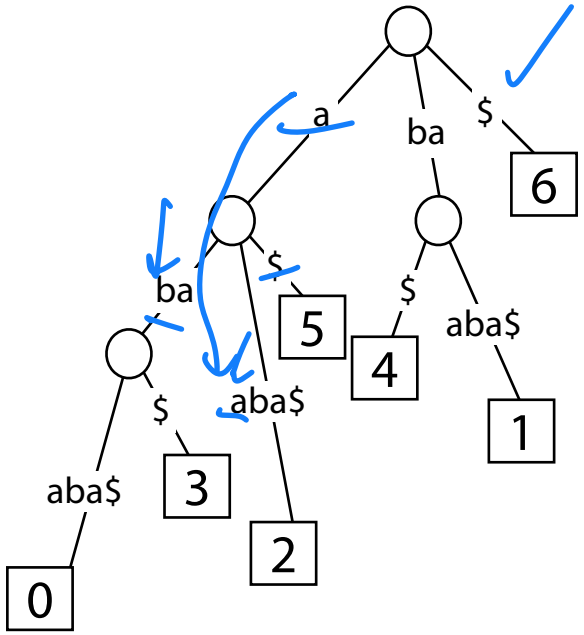


Suffix array: build by suffix tree

up to alphabet degree

(a) Build suffix tree, (b) traverse in lexicographic order, (c) upon reaching leaf, append suffix to array

1) Find smallest char



6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

Assignment 7: a_sarray

Learning Objective:

Construct a suffix array by sorting suffixes

Implement exact pattern matching using a suffix array

Be as efficient or inefficient as you like!

Challenge yourself: Try to build in $O(m^2 \log m)$ or better.

Searching a suffix array

To find all exact matches using a suffix array:

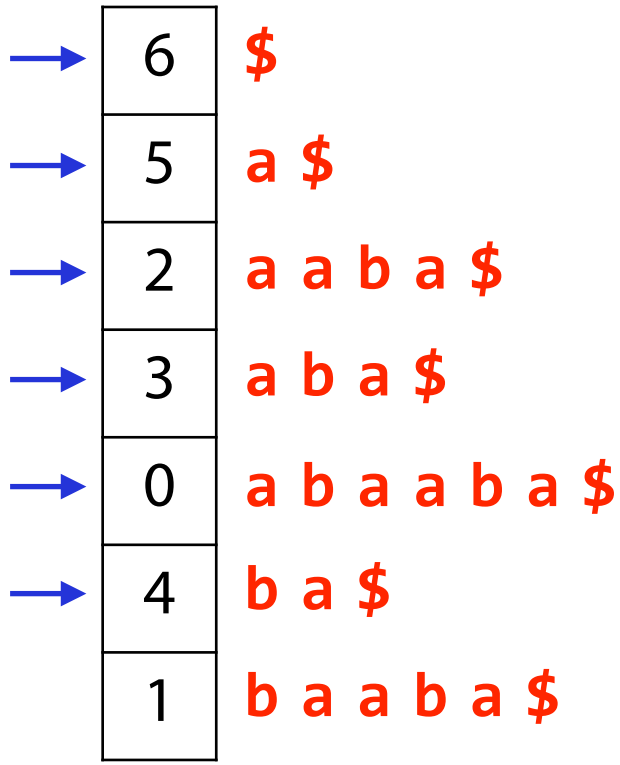
1) Rebuild string

2) Match character

T = abaaba$

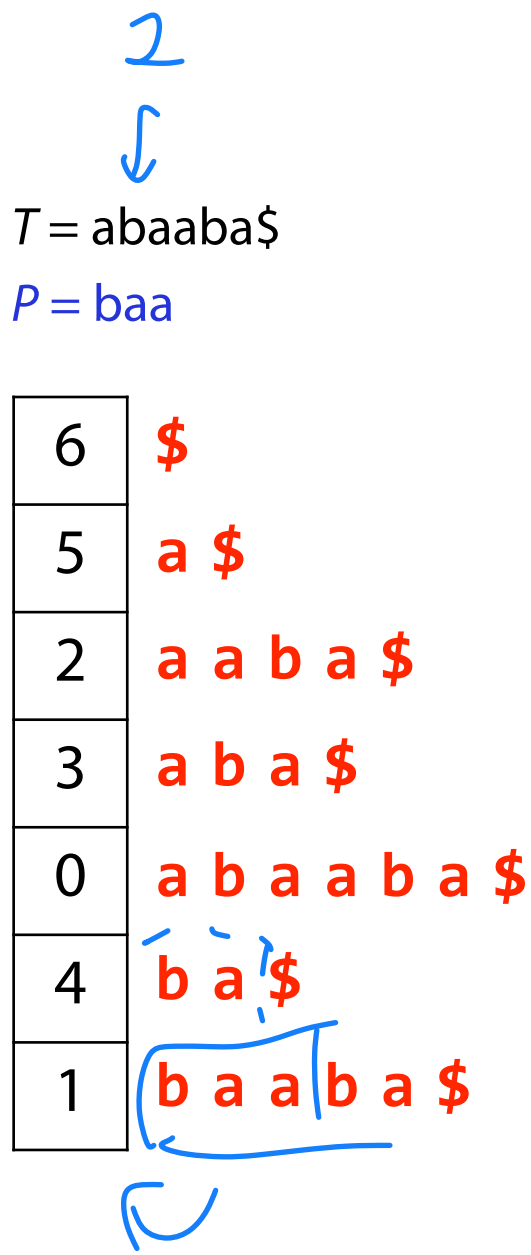
P = baa

Starts with b?



Searching a suffix array

To find all exact matches using a suffix array:

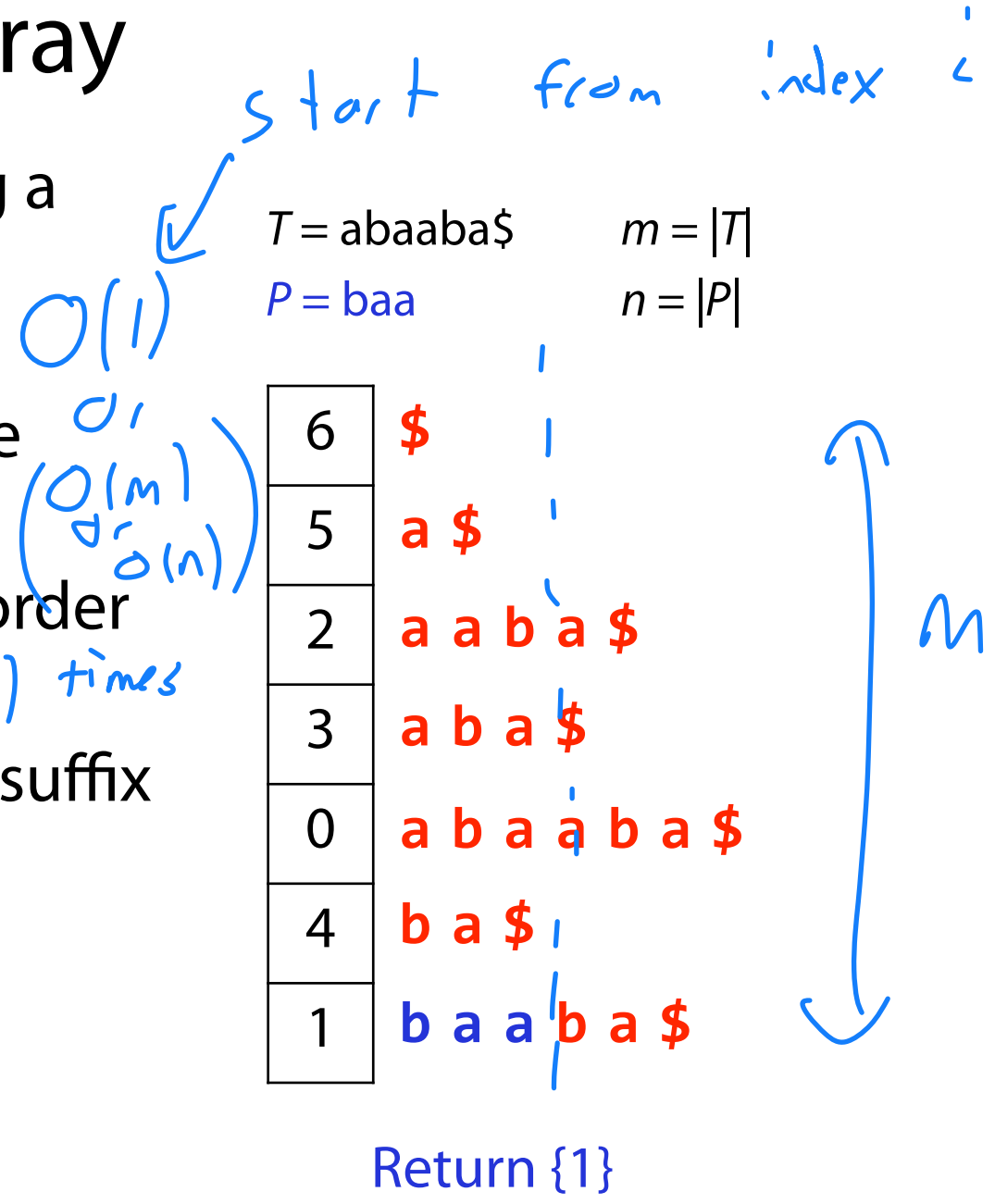


Searching a suffix array

To find all exact matches using a suffix array:

1. Recreate suffix from int value
2. Compare each character in order
3. On mismatch, move to next suffix

What is our time complexity?

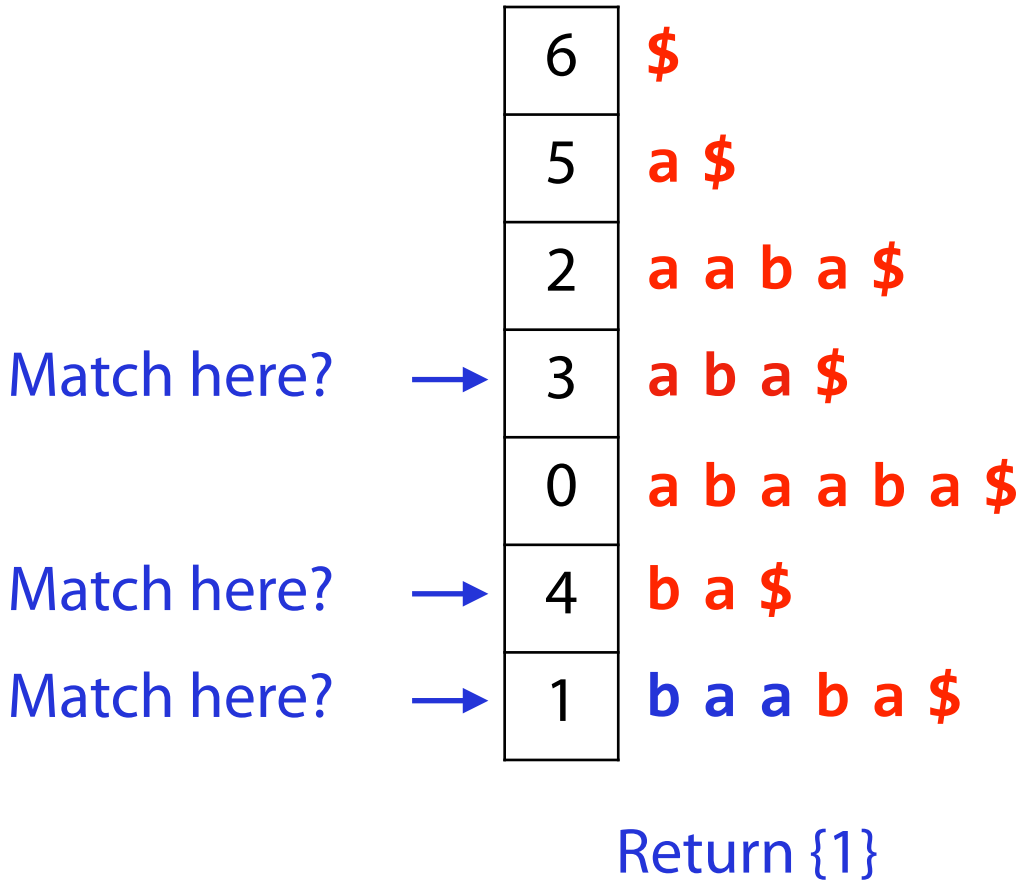


Searching a suffix array

To find all exact matches using a suffix array w/ binary search:

T = abaaba$ $m = |T|$

P = baa $n = |P|$



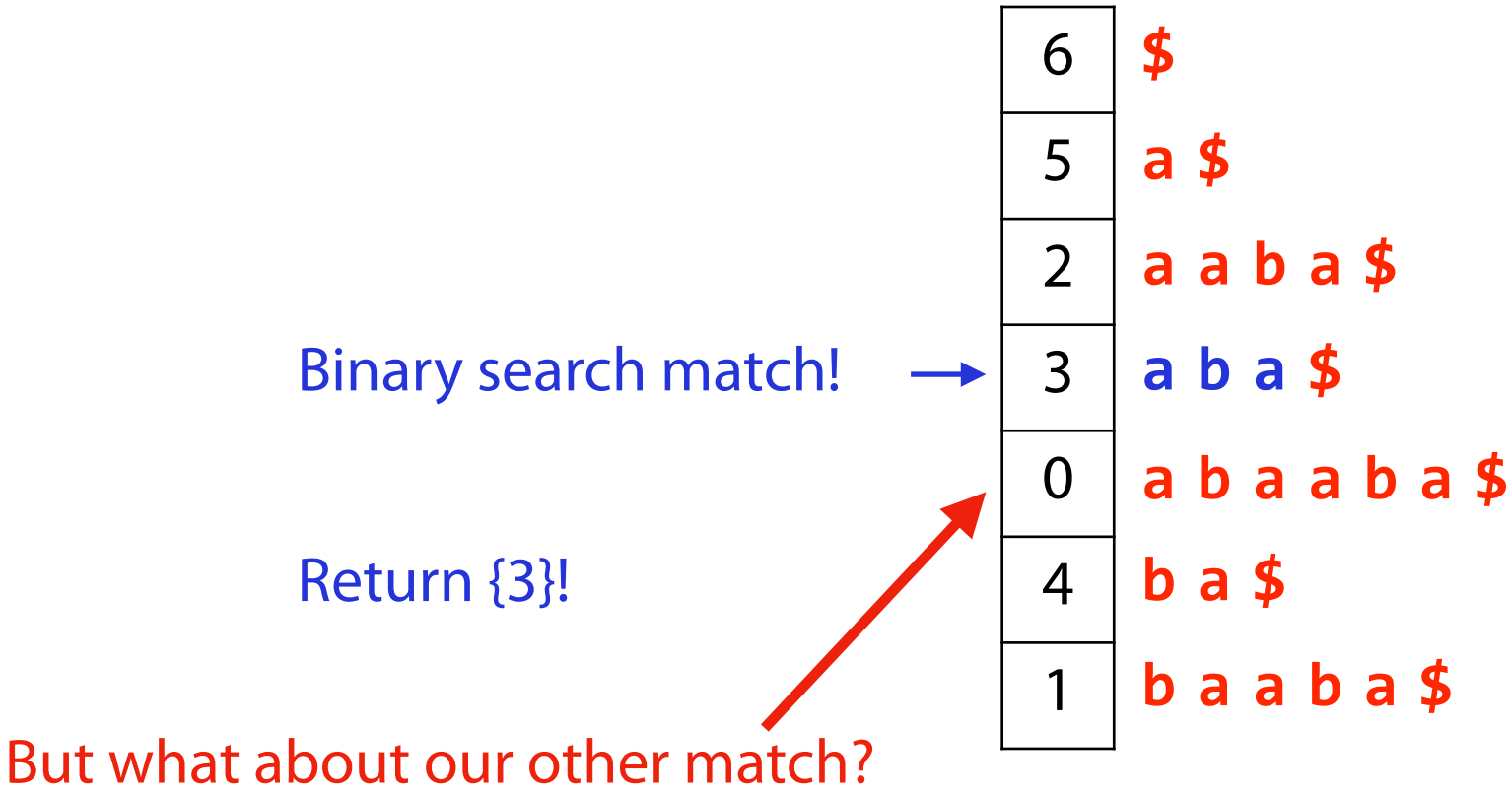
Return {1}

Searching a suffix array

To find all exact matches using a suffix array w/ binary search:

T = abaaba$ $m = |T|$

P = aba $n = |P|$



Searching a suffix array

To find all exact matches using a suffix array w/ binary search:

1. Pick suffixes using binary search
   ↳ $O(n \log m)$
2. Compare suffixes as normal
3. After match, check neighbors
   $O(m * n)$

Assume we have $k=m$ matches

What is our time complexity?

$O(n \log m + mn)$

T = abaaba$ $m = |T|$

P = aba $n = |P|$

6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

No match

Match

No match

Return {0,3}

Searching a suffix array

How can we do better?

Search for first & last

T = abaaba$ $m = |T|$

P = a $n = |P|$

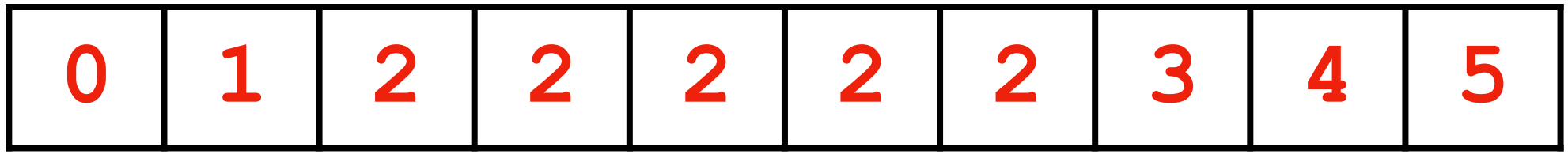
6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

Range Search

Given a collection of objects, C, with comparable values and an object of interest, q, find ~~the first~~ instance(s) of $q \in C$.

ALL

Input:

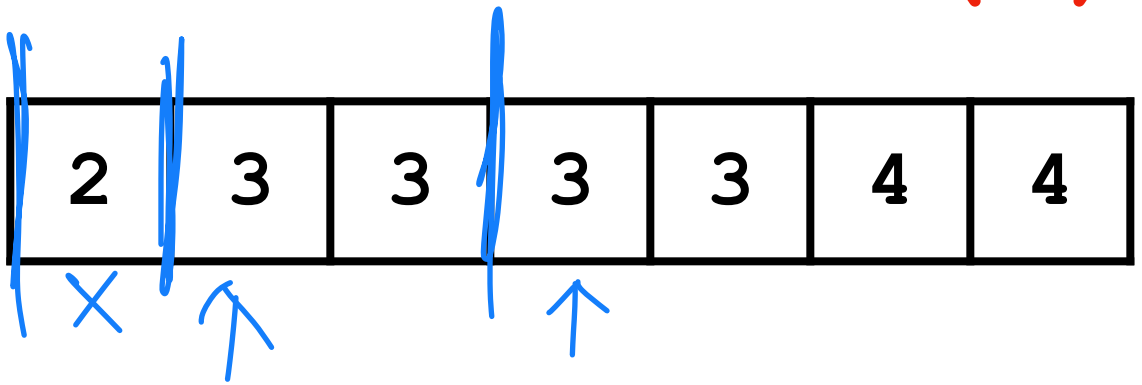


Output: Range of indices matching q if it exists, $(-1, -1)$ otherwise

Binary Search: Get first match

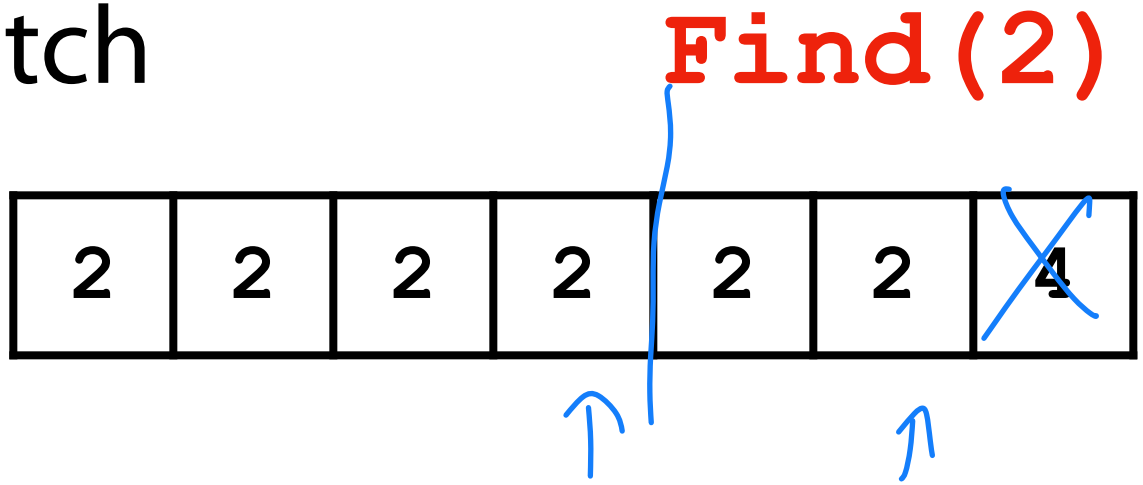
Find(3)

```
1
2           if mid == q:
3
4               # Match case:
5               # Treat like query is larger
6               # Remember last match!
7
8           elif mid > q:
9
10              # query is smaller case
11          else:
12
13              # query is larger case
14
15
16      # Final Return Snippet
17      if saw_match:
18          return last_match
19      else:
20          return -1
21
22
23
```



Binary Search: Get last match

```
1
2           if mid == q:
3
4               # Match case:
5               # Treat like query is smaller
6               # Remember last match!
7
8           elif mid > q:
9
10              # query is smaller case
11          else:
12
13              # query is larger case
14
15
16      # Final Return Snippet
17      if saw_match:
18          return last_match
19      else:
20          return -1
21
22
23
```



Searching a suffix array

How can we do better?

1. Identify the *first* and *last* matches to P w/ binary search
2. Return all values in that range!

Assume we have $k=m$ matches

What is our time complexity?

T = abaaba$ $m = |T|$

P = a $n = |P|$

6	$	
5	a $	First
2	a a b a $	
3	a b a $	
0	a b a a b a $	Last
4	b a $	
1	b a a b a $	

Assignment 7: a_sarray

Learning Objective:

Construct a suffix array by sorting suffixes

Implement exact pattern matching using a suffix array

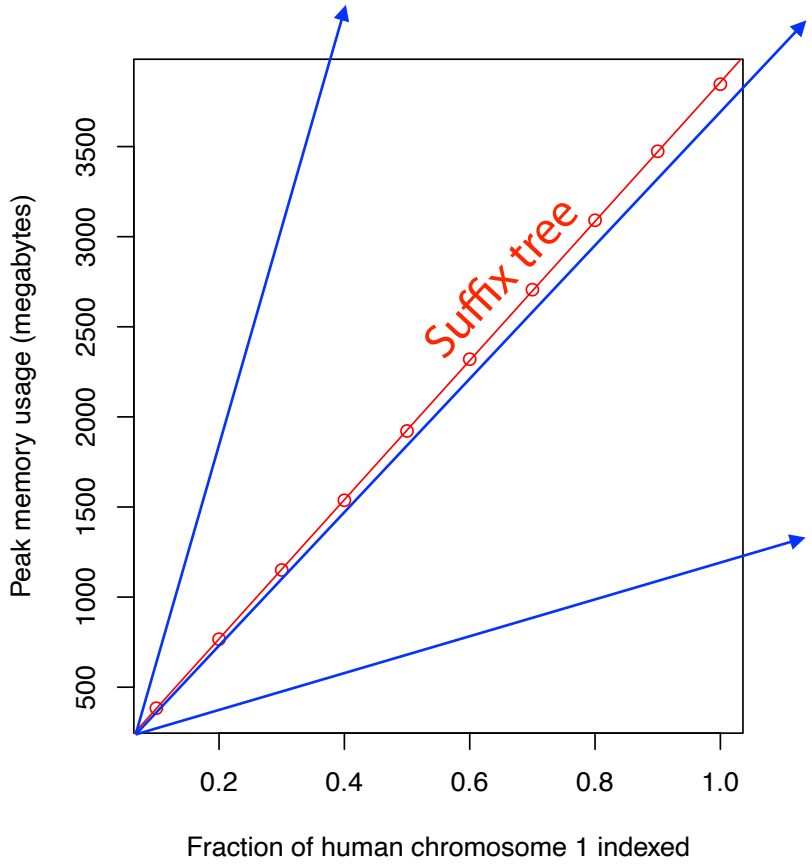
Be as efficient or inefficient as you like!

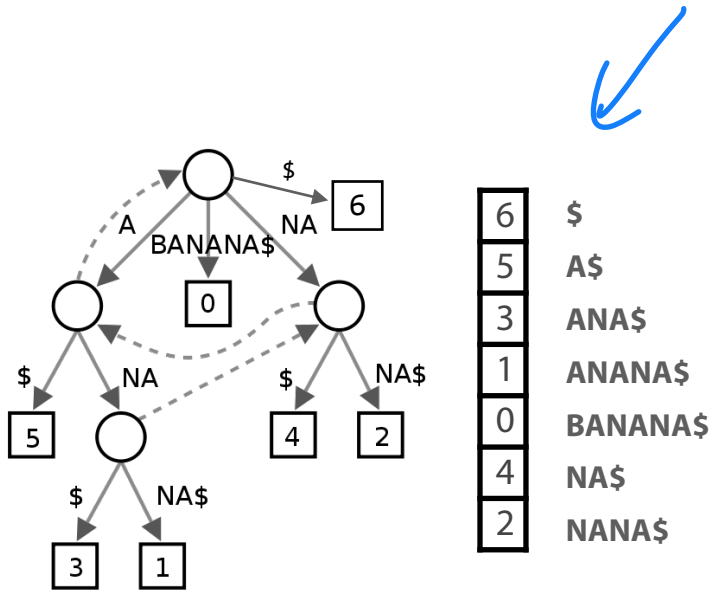
Challenge yourself: Try to search in O(n log m + k)

Suffix tree vs suffix array: size

O(*m*) space, like suffix tree

Is "constant factor" worse, better, same?

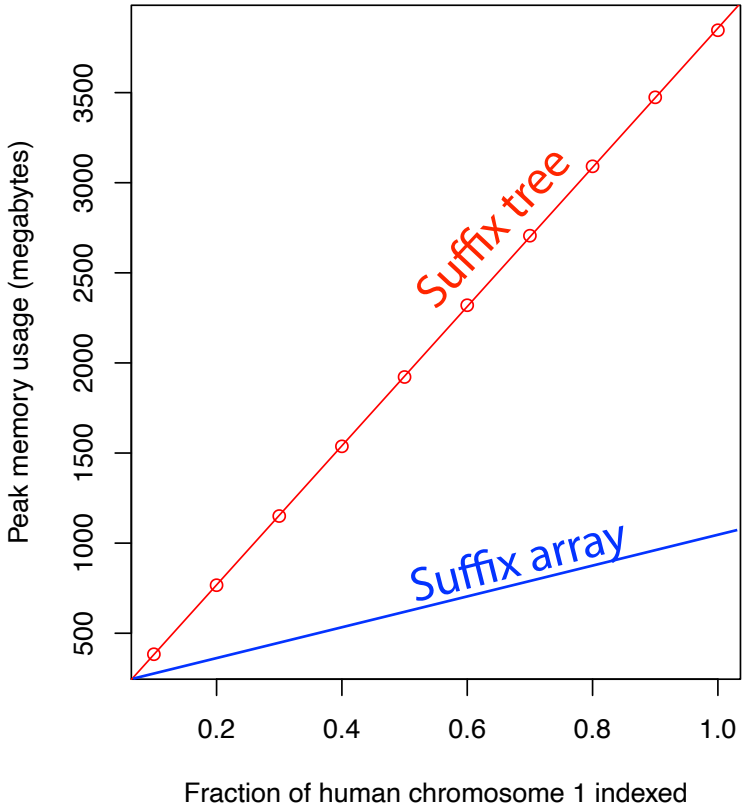


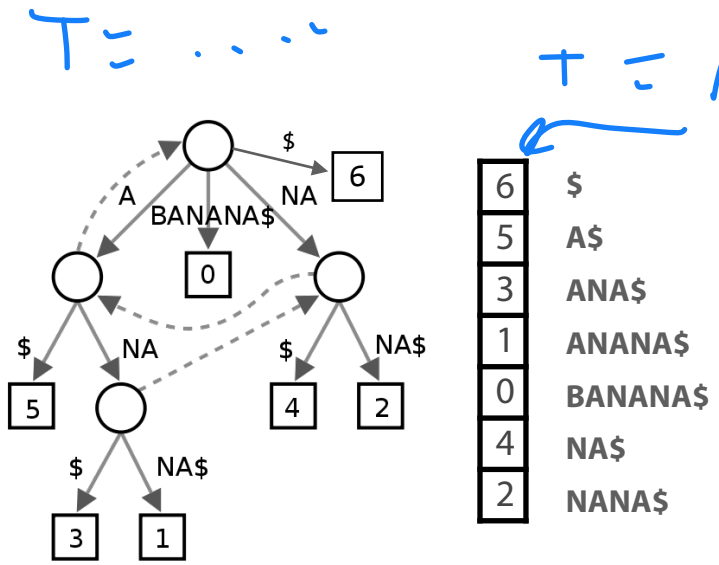


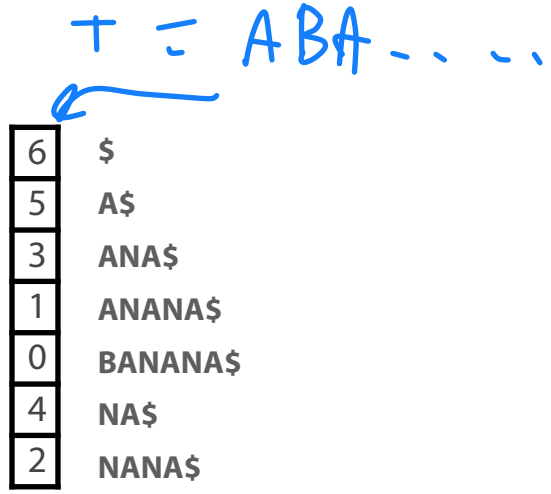
6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

Suffix tree vs suffix array: size

32-bit integers sufficient for human genome, so fits in
~4 bytes/base × 3 billion bases ≈ **12 GB**. Suffix tree is **>45 GB**.







Suffix tree vs suffix array: search

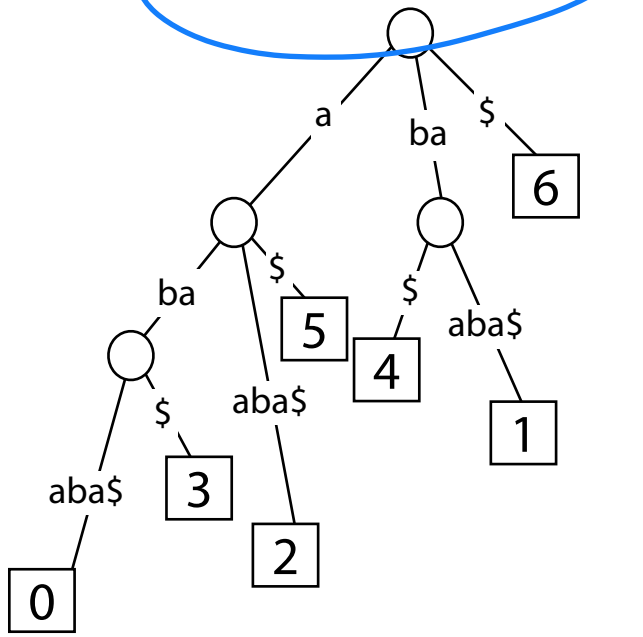
Suffix Array

O($n \log m + k$)*

6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

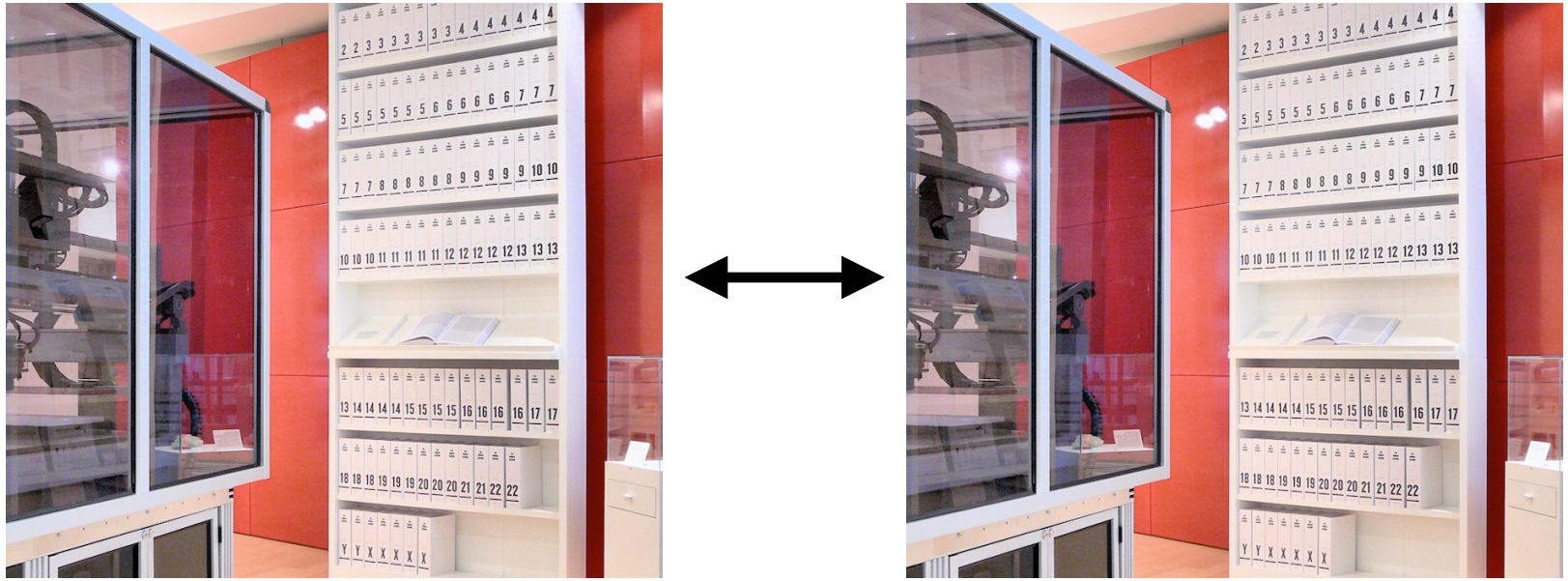
Suffix Tree

O($n + k$)



* Can be improved to O(n + log m), (See Gusfield 7.17.4)

Suffix ***arrays*** in the real world: MUMmer



Delcher, Arthur L., et al. "Alignment of whole genomes." *Nucleic Acids Research* 27.11 (1999): 2369-2376.

Delcher, Arthur L., et al. "Fast algorithms for large-scale genome alignment and comparison." *Nucleic Acids Research* 30.11 (2002): 2478-2483.

Kurtz, Stefan, et al. "Versatile and open software for comparing large genomes." *Genome Biol* 5.2 (2004): R12.

G. Marçais et al. "MUMmer4: A fast and versatile genome alignment system." *PLoS Comp Biol* (2018)

~ 4,000 citations

http://mummer.sourceforge.net

Exact pattern matching *w/ indexing*

There are many data structures built on ***suffixes***

The FM index is a compressed self-index (smaller* than original text)!