Exact String Matching and searching for SNPs

CMSC423 Fall 2014
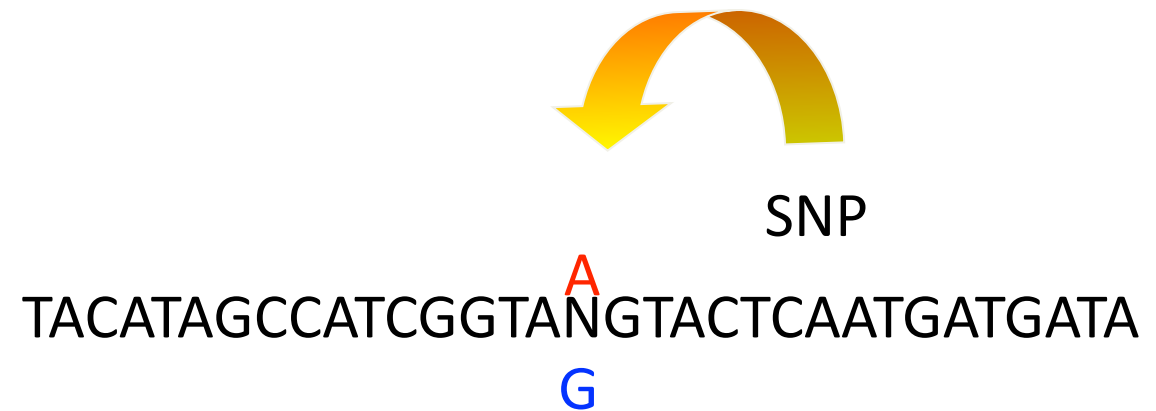
What makes them different?

Much human variation is due to difference in ~ 6 million base pairs (0.1 % of genome) referred to as SNPs

Single Nucleotide Polymorphism (SNP)

SNP

Genomic DNA:

A
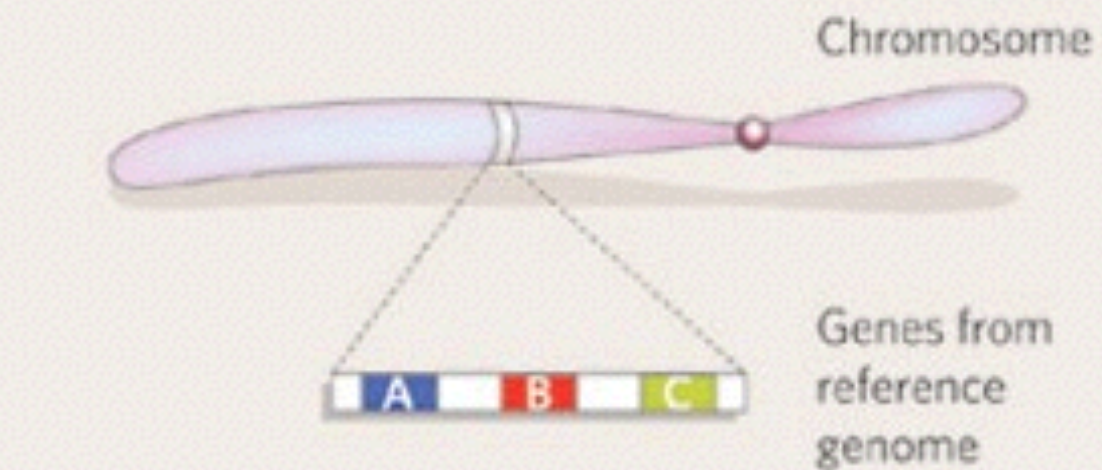TACATAGCCATCGGTANGTACTCAATGATGATA
G

Three genotypes

AA

Mother

TACATAGCCATCGGTA**A**GTACTCAATGATGATA
ATGTATCGGTAGCCAT**T**CATGAGTTACTACTAT

Father

TACATAGCCATCGGTA**A**GTACTCAATGATGATA
ATGTATCGGTAGCCAT**T**CATGAGTTACTACTAT

AG

Mother

TACATAGCCATCGGTAAGTACTCAATGATGATA
ATGTATCGGTAGCCATTCATGAGTTACTACTAT

Father

TACATAGCCATCGGTAGGTACTCAATGATGATA
ATGTATCGGTAGCCATCCATGAGTTACTACTAT

GG

Mother

TACATAGCCATCGGTA**G**GTACTCAATGATGATA
ATGTATCGGTAGCCAT**C**CATGAGTTACTACTAT

Father

TACATAGCCATCGGTA**G**GTACTCAATGATGATA
ATGTATCGGTAGCCAT**C**CATGAGTTACTACTAT

[Check, Nature 437]

Health or Disease?

DNA Sequence

Person 1   **A A A T T T**   **Normal protein**

Some DNA variations have no negative effects

Person 2   **A A T T T T**

Low or nonfunctioning protein

Person 3   **A A C T T T**

Other variations lead to disease (e.g., sickle cell) or increased susceptibility to disease (e.g., lung cancer)

YGG-00-0480

Personal Genomics

Next-gen Sequencing

Platforms

Illumina/Solexa
Solid-phase amplification
One DNA molecule per cluster

Sample preparation
DNA (5 µg)

Template
dNTPs
and
polymerase

Bridge amplification

Source: Metzker ML. Sequencing technologies

sequence
quality scores

x 100s of
millions

A C G T A C G T A C G T A C G T A C G T ... A C G T
1   2   3   4   5   34

Source: Whiteford et al. Swift: primary data analysis for the Illumina Solexa sequencing platform. Bioinformatics. 2009

13

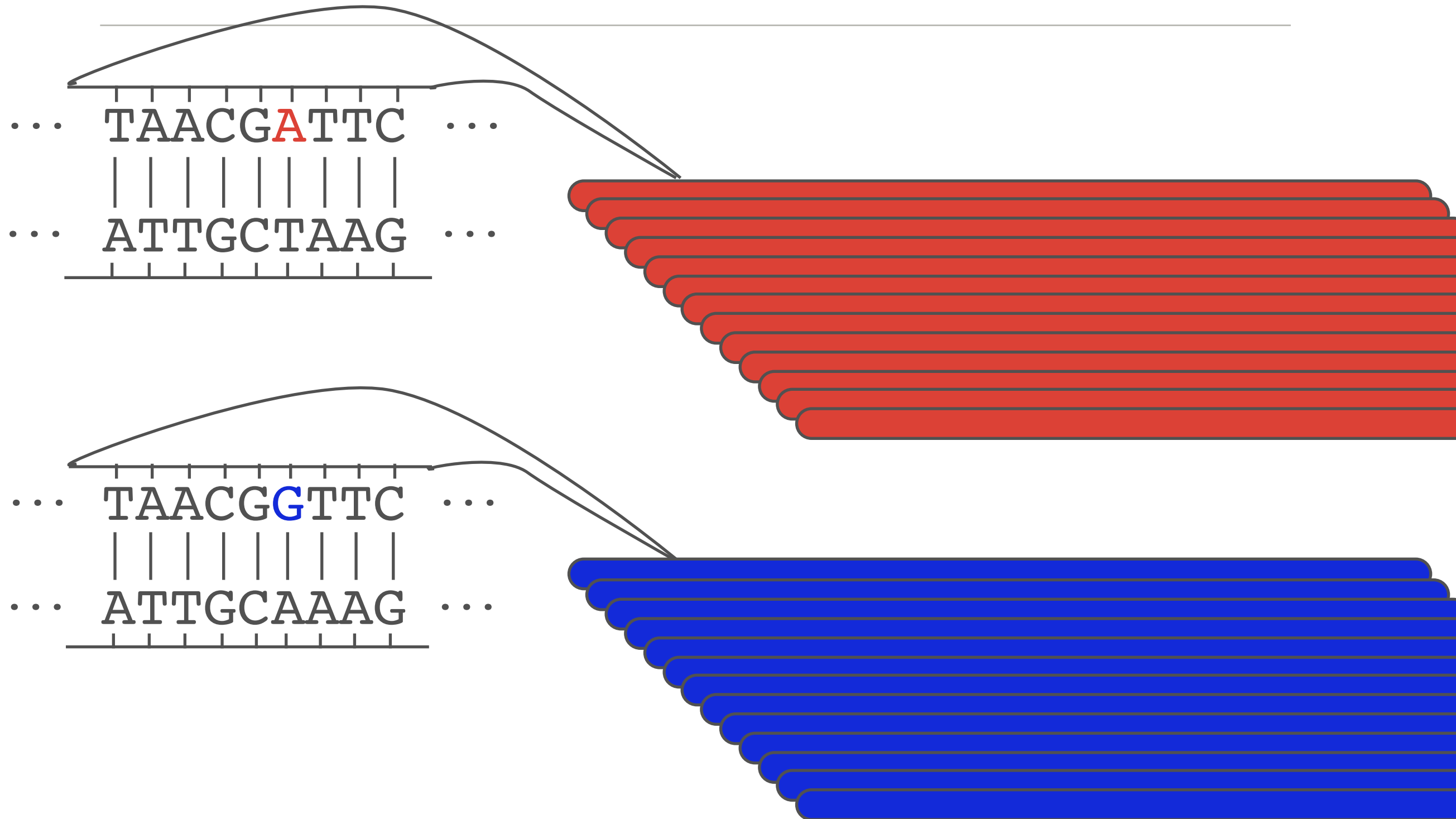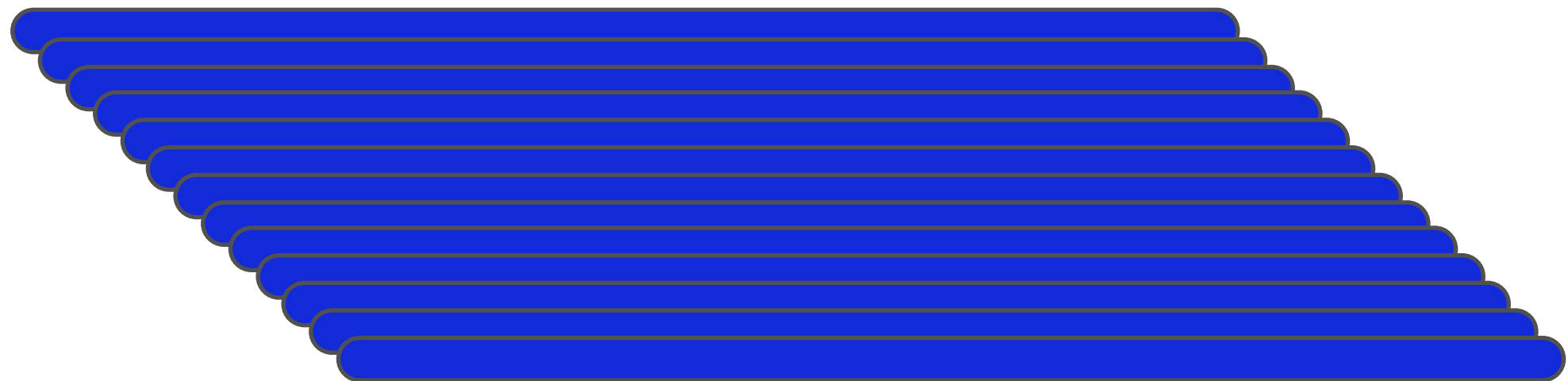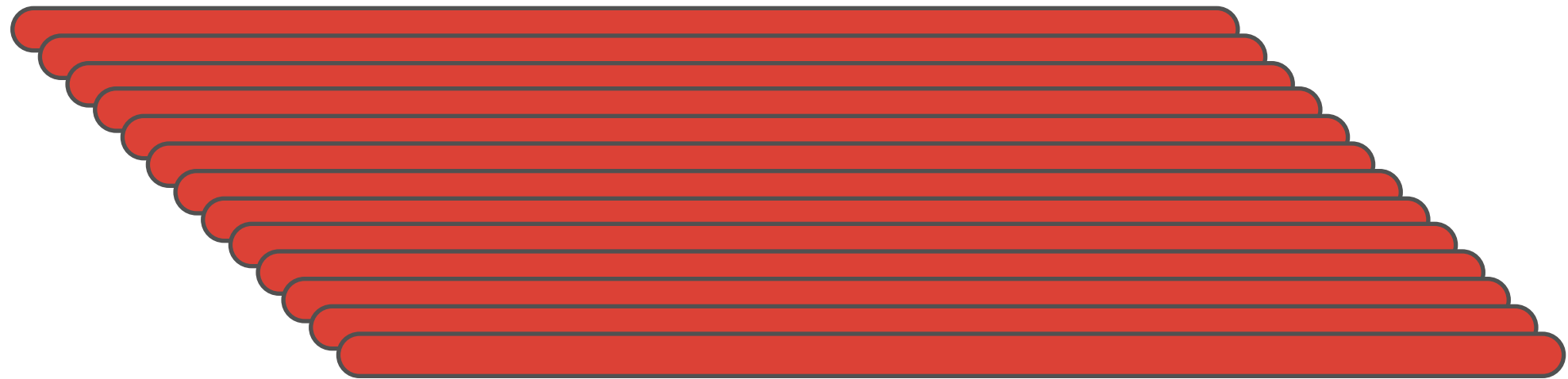cing throughput

Sequencing throu
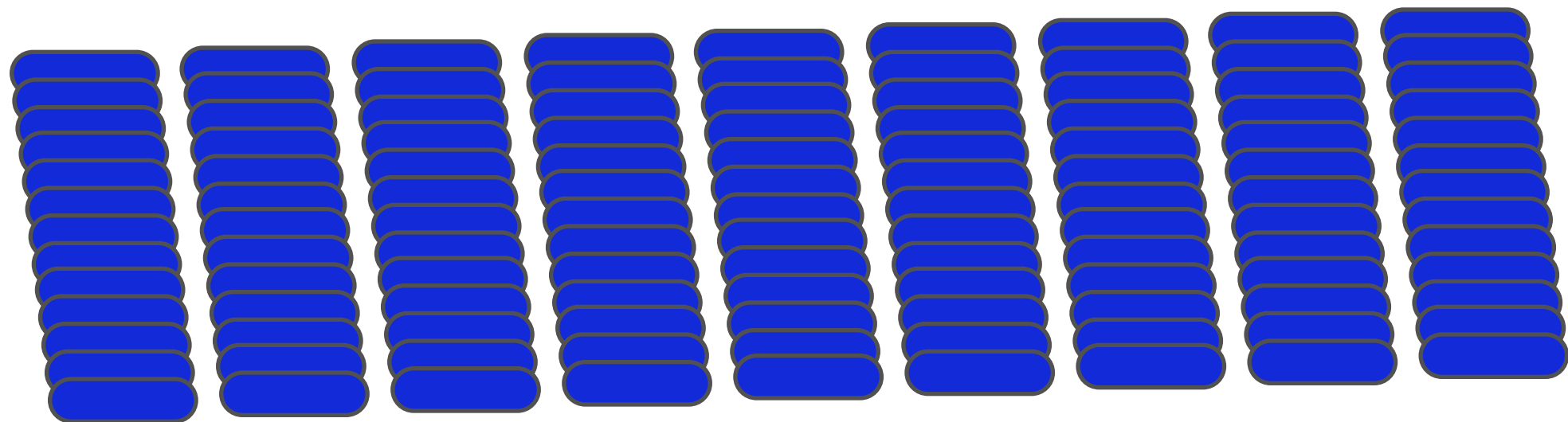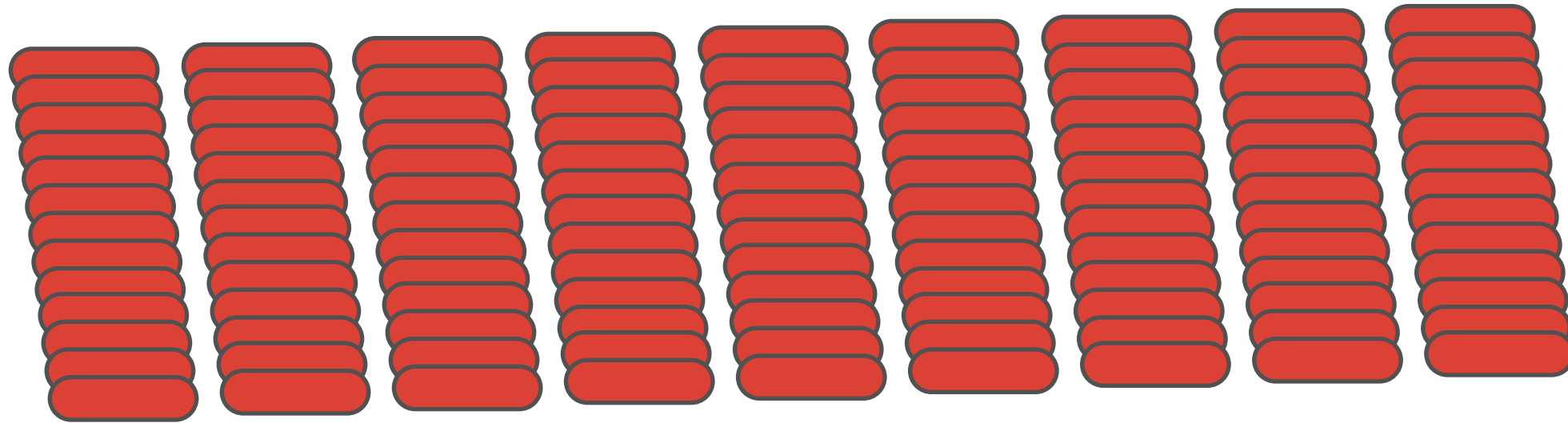
Sec-gen Sequencing for SNPs

Sec-gen Sequencing for SNPs

Sec-gen Sequencing for SNPs

Sec-gen Sequencing for SNPs

Sec-gen Sequencing for SNPs

"Pileup" or "Coverage plot"

GTCGCAGTA**N**CTGTCT
||||||||| ||||||
GTCGCAGTA**T**CTGTCT

GGATCT**G**CGATATACC
|||||| |||||||||
GGATCT–CGATATACC

AATCTGATCTTATTTT
||||||||||||||||
AATCTGATCTTATTTT

ATATATATATATATAT
||||||||||||||||
ATATATATATATATAT

TCTCTCCCA**NN**AGAGC
||||||||| ||||||
TCTCTCCCA**GG**AGAGC

Align

Aggregate

"Depth of coverage" = 14

Statistics

Call: HET A, G
p-value: 0.0023

GTCGCAGTATCTG**A**CT
GTCGCAGTATCTGT**NN**
TGTCGCAGTATCTGTC
TAT**G**TCGCAGTATCTG
TAT**A**TCGCAGTATCT**T**
TATATCGCAGTATCTG
NATATCGCAGTAT**N**TG
CCCTATATCGCAGTAT
ACACCCTAT**G**TCGCA
ACACCCTAT**C**TCGCA
ACACCCTAT**G**TCGCA
GA–CACCCTAT**G**TCGC
CCGGA–CACCCTAT**A**T
CCGGA–CACCCTAT**A**T
GCCGGA–CACCCTAT**G**

"Coverage"

TGGTATTTTCGTCTGGGGGGTATGCACGCGATAGCATTGCGAGACGCTGGAGCCGGAGCACCCTAT**G**TCGCAGTATCTGTCTTTGATTCCTGCCTCATCCTATTAT

Reference

(slide courtesy of Ben Langmead)

The problem

- Given:
  - 100's of millions of short reads: 100-200bp reads
  - A long reference genome (~3Bbp for human)
- Do:
  - Find high scoring scoring (fitting) alignments for each read
- What we know:
  - Dynamic programming solution for fitting alignment:
    - 1e8 * 1e9 * 1e2 operations, 1e9 * 1e2 memory

Strategies

- What if we only allow a small number of substitutions?
  - Let's first try to find *exact* matches and work from those (the d+1 trick in the midterm)
- We are aligning to the same reference 100's of millions of time
  - Is there preprocessing we can do to amortize time?
- Genomes are repetitive
  - Can we search for matches in the genome in a smart way?
  - Can we compress the genome, and search over the compressed representation?

Exact Matching

- Suffix Tries/Trees
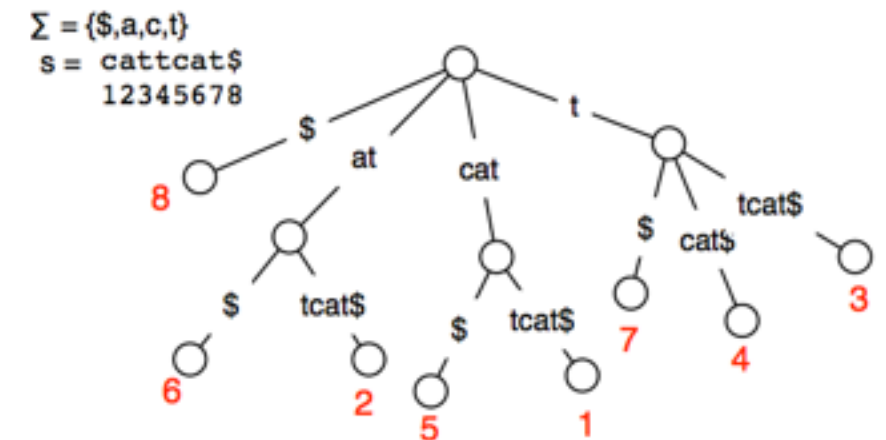
- Suffix Arrays
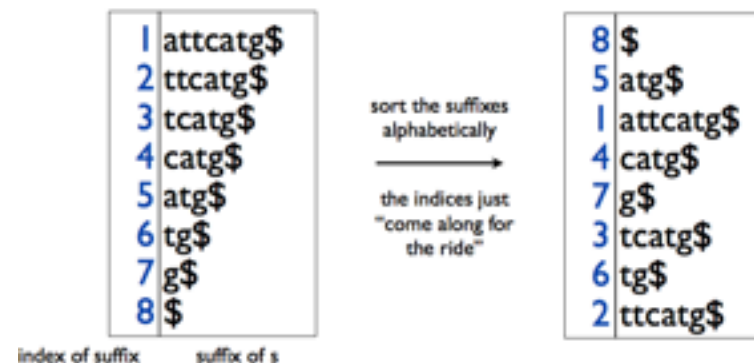
- The Burrows-Wheeler transform

State of the Art

- Bowtie: ultra-fast mapping of short reads to reference genome

Bowtie
An ultrafast memory-efficient short read aligner

Bowtie is an ultrafast, memory-efficient short read aligner. It aligns short DNA sequences (reads) to the human genome at a million 35-bp reads per hour. Bowtie indexes the genome with a Burrows-Wheeler index to keep its memory footprint small: GB for the human genome (2.9 GB for paired-end).

- http://bowtie-bio.sourceforge.net

Suffix Trie
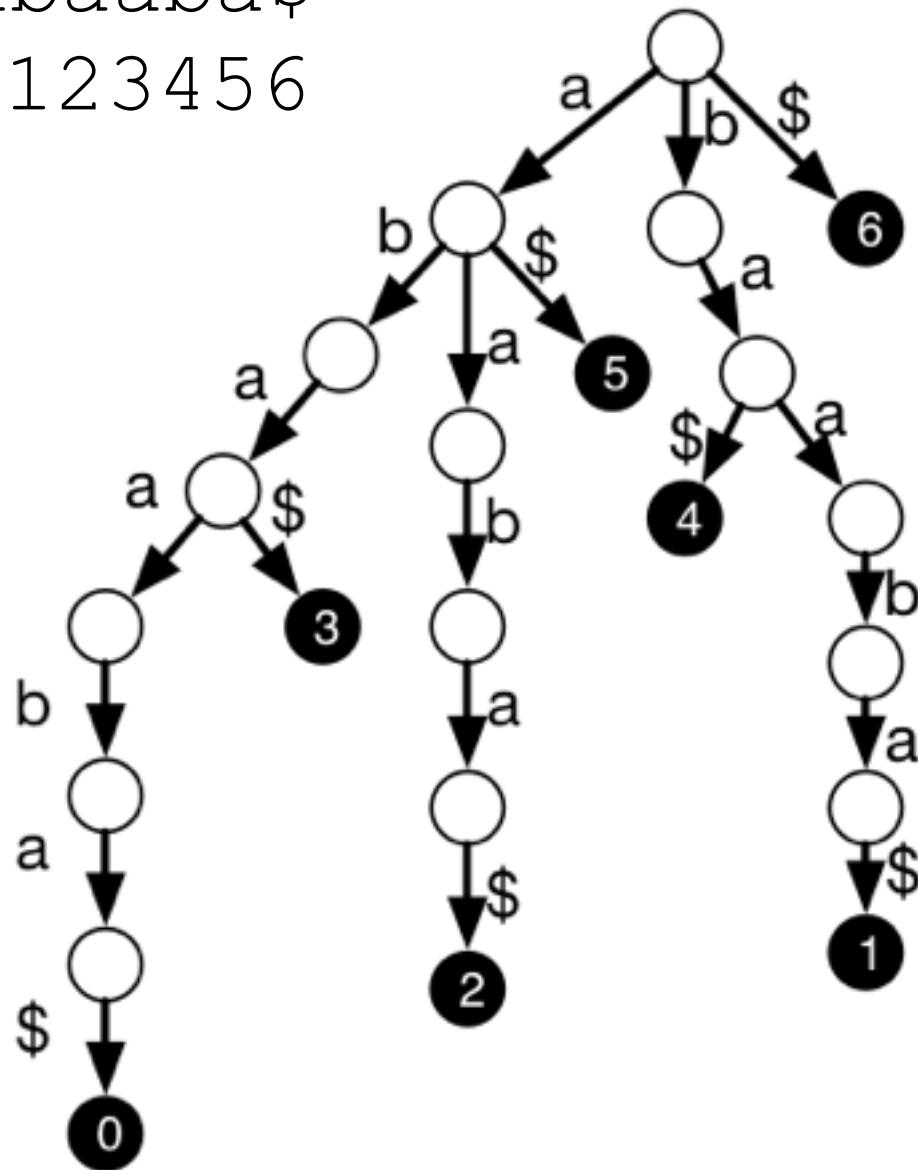
`T: abaaba$`
`   0123456`

- Edges labeled with characters from alphabet.

- Each path from root to leaf corresponds to a *suffix* of T.

Suffix Trie

T: abaaba$
   0123456
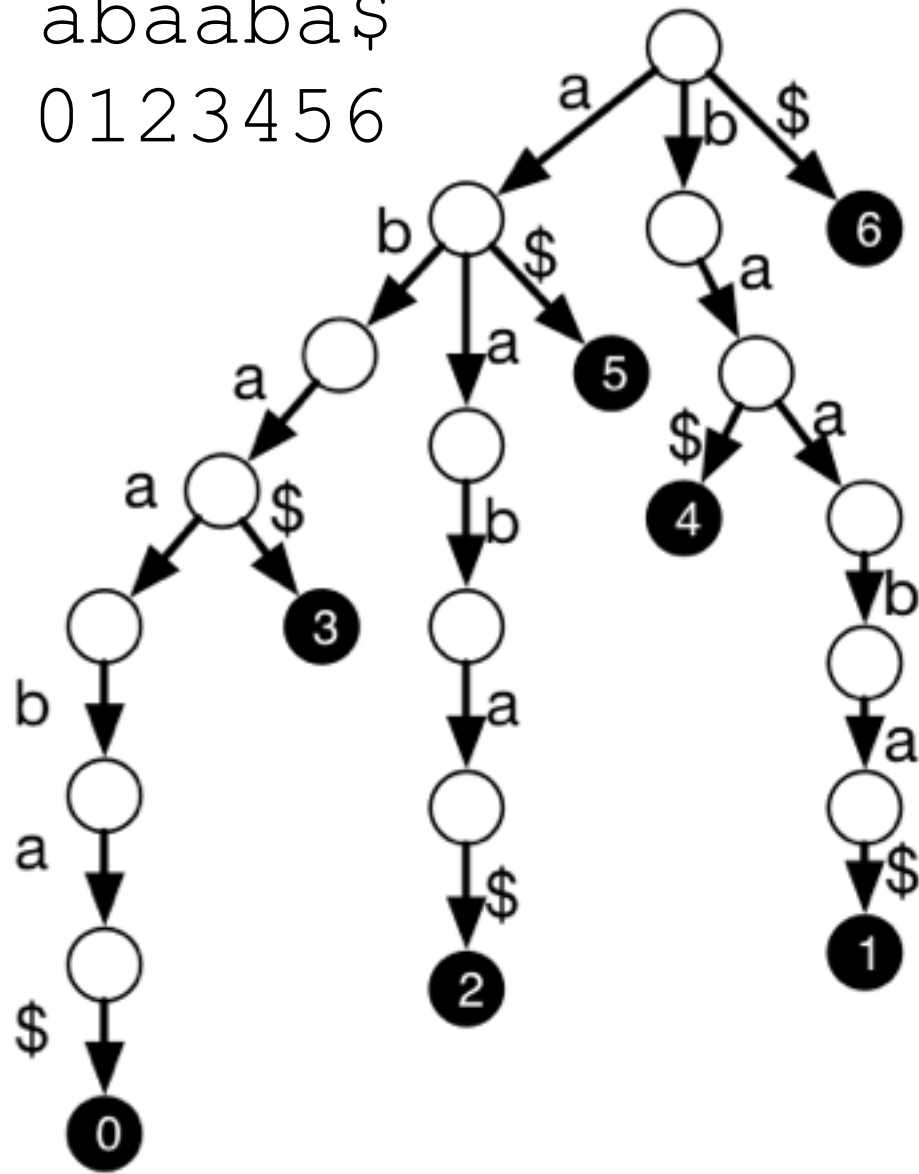
- Naive construction algorithm is $O(|T|^2)$
  - There are linear time construction algorithms (see Gusfield)

- Memory requirement is also $O(|T|^2)$

- Time to find matches $O(|P|)$

Suffix Tree

T: abaaba$
   0123456
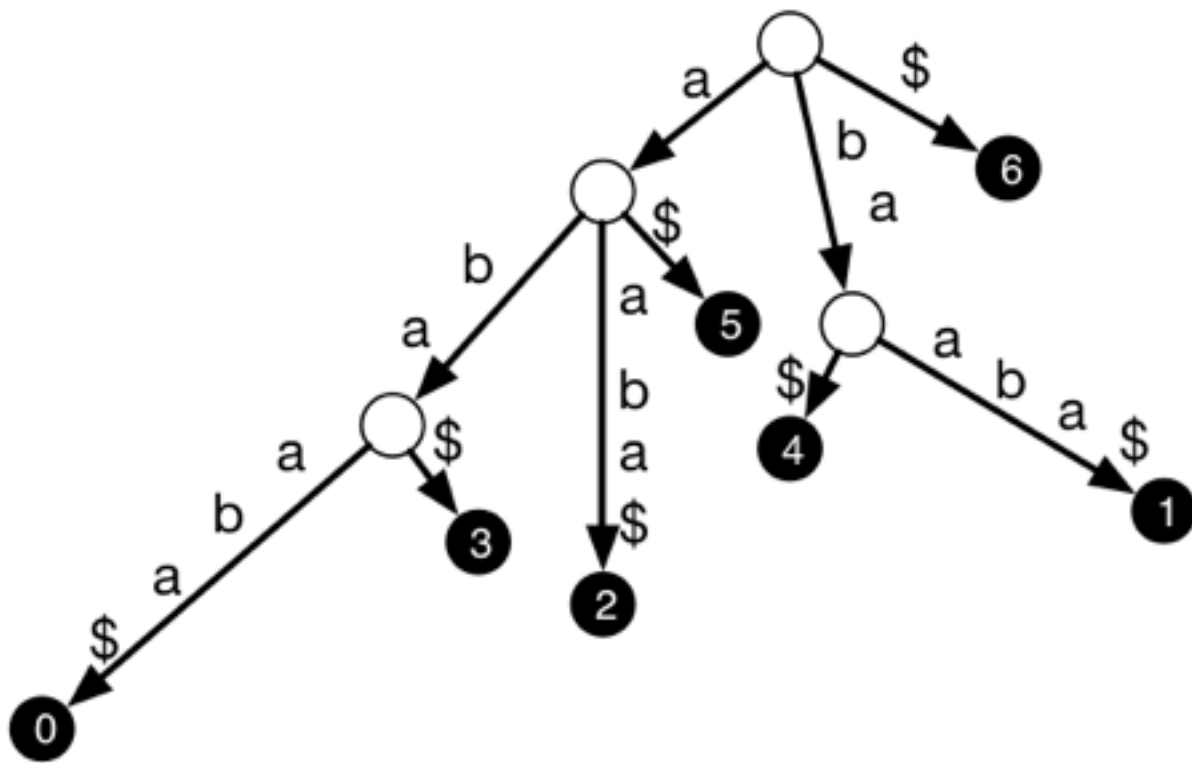
- Collapse non-branching nodes
  - #nodes O(ITI)

- Memory requirement is *not* O(ITI)
  - In the worst case, space required for edge labels is O(ITI)

Suffix Tree

T: abaaba$
   0123456
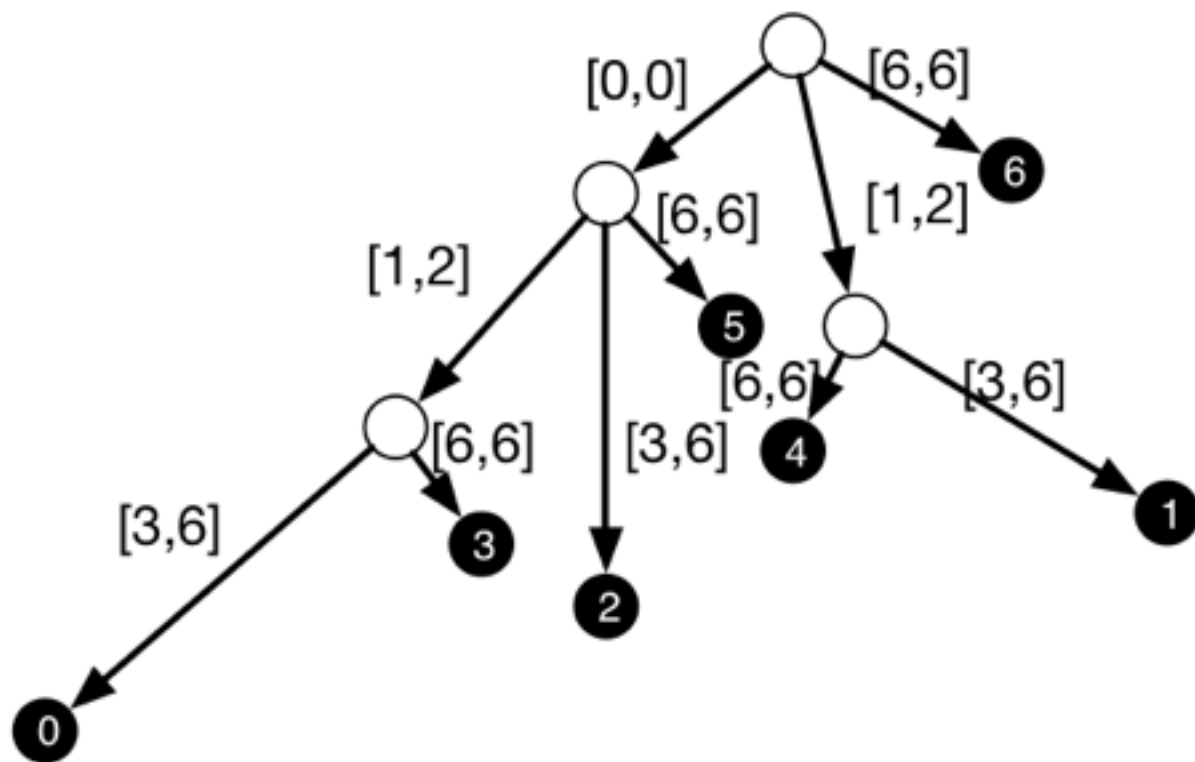
- Collapse non-branching nodes
  - #nodes O(|T|)

- Label edges with substring *[start,end]*
  - O(1) per edge

- Memory now O(|T|)

- Construction algorithm O(|T|) (see Gusfield)

Suffix Arrays

- Even though Suffix Trees are O(n) space, the constant hidden by the big-Oh notation is somewhat "big": ≈ 20 bytes / character in good implementations.

- If you have a 10Gb genome, 20 bytes / character = 200Gb to store your suffix tree. "Linear" but large.

- Suffix arrays are a more efficient way to store the suffixes that can do most of what suffix trees can do, but just a bit slower.

- Slight space vs. time tradeoff.

Example Suffix Array

s = attcatg$

- Idea: lexicographically sort all the suffixes.

- Store the starting indices of the suffixes in an array.

index of suffix	suffix of s
1	attcatg$
2	ttcatg$
3	tcatg$
4	catg$
5	atg$
6	tg$
7	g$
8	$

sort the suffixes alphabetically

→

the indices just "come along for the ride"

8	$
5	atg$
1	attcatg$
4	catg$
7	g$
3	tcatg$
6	tg$
2	ttcatg$

Example Suffix Array

- Idea: lexicographically sort all the suffixes.

- Store the starting indices of the suffixes in an array.

s = attcatg$

index of suffix	suffix of s
1	attcatg$
2	ttcatg$
3	tcatg$
4	catg$
5	atg$
6	tg$
7	g$
8	$

sort the suffixes alphabetically

the indices just "come along for the ride"

8
5
1
4
7
3
6
2

Another Example Suffix Array

s = cattcat$

- Idea: lexicographically sort all the suffixes.

- Store the starting indices of the suffixes in an array.

index of suffix	suffix of s
1	cattcat$
2	attcat$
3	ttcat$
4	tcat$
5	cat$
6	at$
7	t$
8	$

sort the suffixes alphabetically

the indices just "come along for the ride"

8	$
6	at$
2	attcat$
5	cat$
1	cattcat$
7	t$
4	tcat$
3	ttcat$

index of suffix   suffix of s

Another Example Suffix Array

s = cattcat$

- Idea: lexicographically sort all the suffixes.

- Store the starting indices of the suffixes in an array.

index of suffix	suffix of s
1	cattcat$
2	attcat$
3	ttcat$
4	tcat$
5	cat$
6	at$
7	t$
8	$

sort the suffixes alphabetically →

the indices just "come along for the ride"

8
6
2
5
1
7
4
3

Search via Suffix Arrays

s = cattcat$

8	$
6	at$
2	attcat$ ← √
5	cat$
1	cattcat$ ←
7	t$
4	tcat$
3	ttcat$

- Does string "at" occur in s?
- Binary search to find "at".

- What about "tt"?

Counting via Suffix Arrays

s = cattcat$

8	$
6	at$
2	attcat$
5	cat$
1	cattcat$
7	t$
4	tcat$
3	ttcat$

- How many times does "at" occur in the string?

- All the suffixes that start with "at" will be next to each other in the array.

- Find one suffix that starts with "at" (using binary search).

- Then count the neighboring sequences that start with at.

Constructing Suffix Arrays

- Easy $O(n^2 \log n)$ algorithm:

    > sort the n suffixes, which takes $O(n \log n)$ comparisons, where each comparison takes $O(n)$.

- There are several direct $O(n)$ algorithms for constructing suffix arrays that use very little space.

- An simple $O(n)$ algorithm: build the suffix tree, and exploit the relationship between suffix trees and suffix arrays (next slide)

Relationship between Suffix Arrays and Suffix Trees

T: abaaba$
   0123456
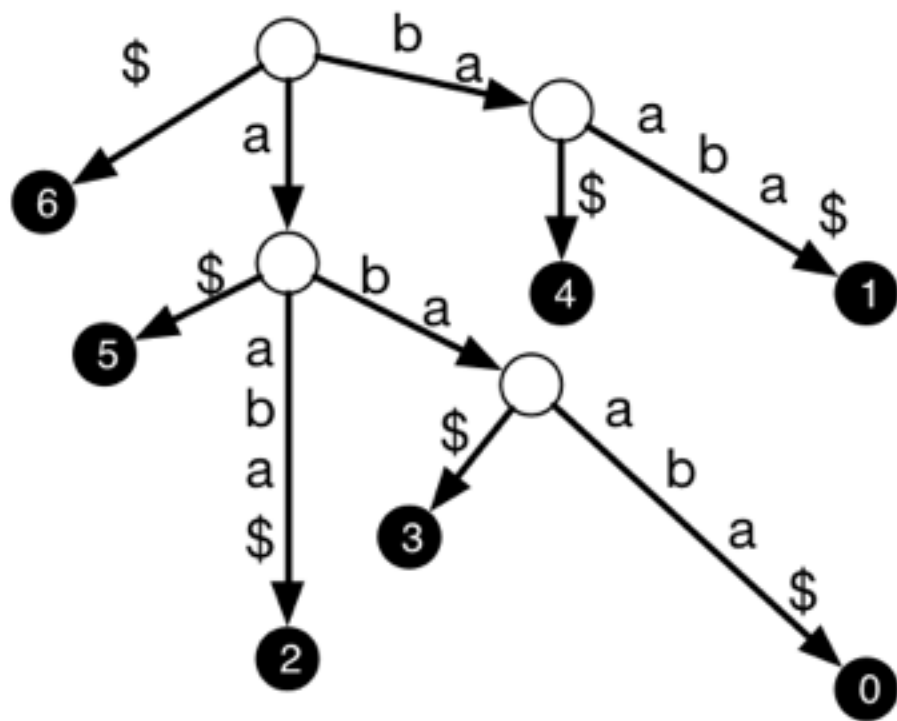
6	$
5	a$
2	aaba$
3	aba$
0	abaaba$
4	ba$
1	baaa$

Build suffix trees with edge labels sorted lexicographically
Order of leaves: 6,5,2,3,0,4,1

Recap

Structure	Processing Time	Memory	Search
Suffix Trie	O(|T|)	O(|T|	O(|P|)
Suffix Tree	O(|T|)	O(|T|)*	O(|P|)
Suffix Array	O(|T|)	O(|T|) (but **much** smaller than Suffix Tree)	O(|P|log

*In best implementations about 20 bytes per character (as opposed to 4 bytes for suffix array)