Practical read mapping

Analysis of High Throughput Sequencing Data
2-14-2017

Indexing, software, bam files

- FM-indexing
  - Burrows-Wheeler transform
  - First-last mapping
  - Noisy matching
- Software
  - Bowtie 2
- Bam Files
  - Format
  - using pysam to parse

Difficulties with strategies we've discussed so far

- String matching
  - Very very slow!
  - Memory efficient
  - Can easily handle mismatches
- Hashing
  - Very fast!
  - Memory inefficient
  - Need to be clever to handle mismatches

What do we want in a mapping algorithm

- Doesn’t need to search the entire genome every time
- Relatively low memory footprint
- Easy handling of mismatches

FM-indexing is a powerful solution

- Permute the genome into a convenient form
  - Burrows-Wheeler transform
- Retain an index to look things up
  - Suffix array
- Allow for mismatches
  - Backtracing

Burrows-Wheeler transform

someword

append terminal character

someword$

generate suffixes

someword$
$someword
d$somewor
rd$somewo
ord$somew
word$some
eword$som
meword$so
omeword$s

Sort lexicographically

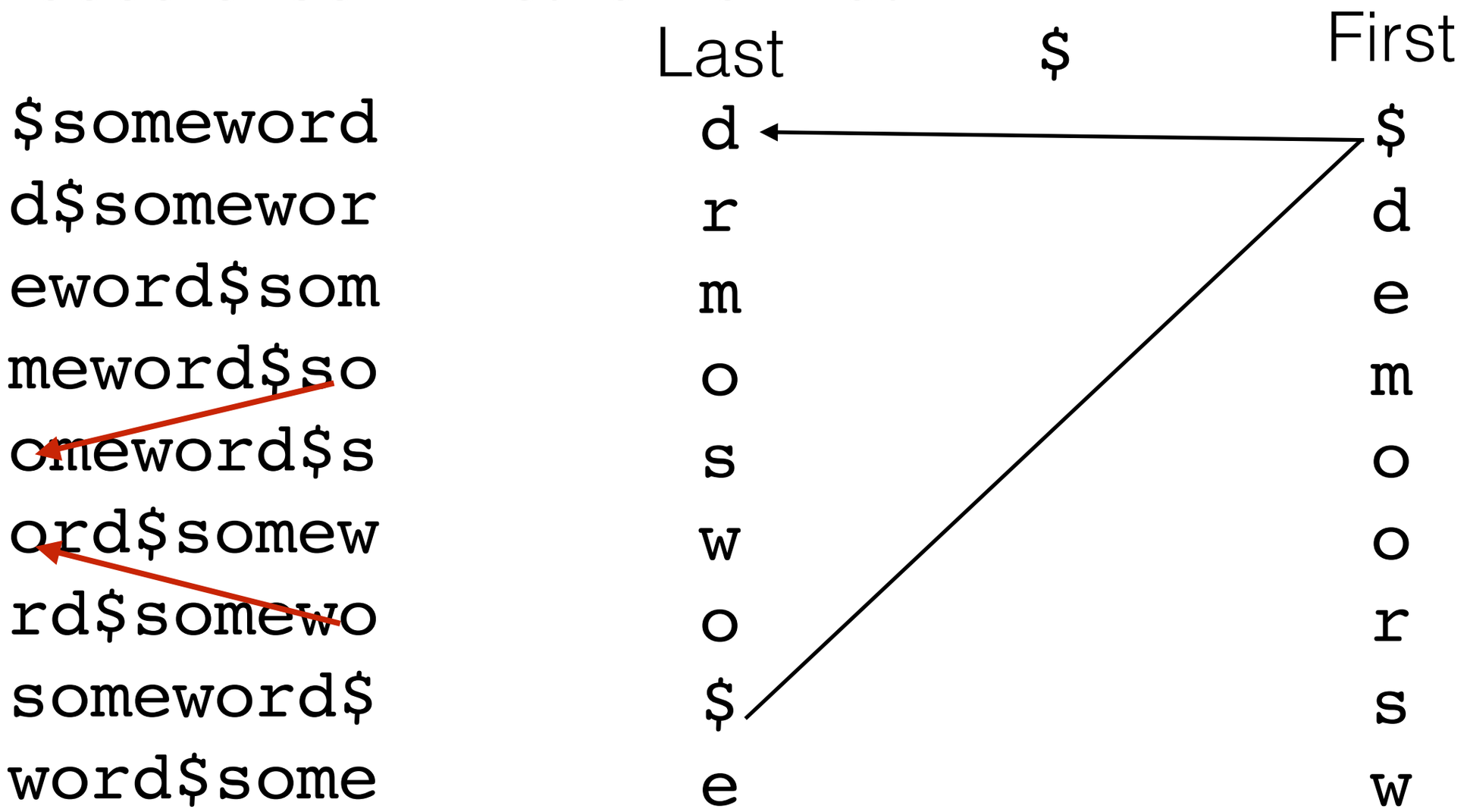
BWT matrix

drmoswo$e
$someword
d$somewor
eword$som
meword$so
omeword$s
ord$somew
rd$somewo
someword$
word$some

Take final column

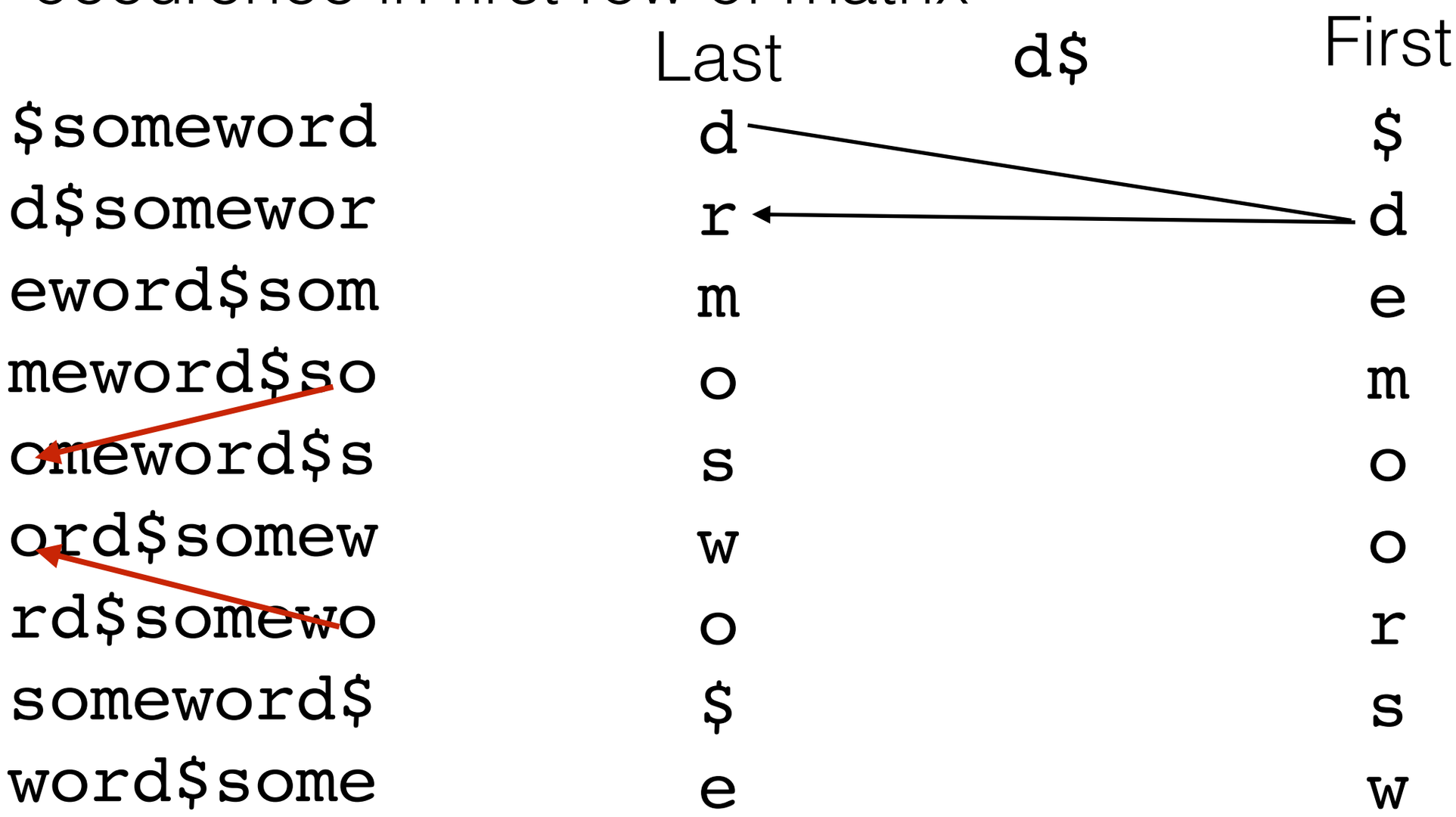
Last-First mapping makes BWT reversible

- The *i*th occurence of a letter in the BWT is same as *i*th occurence in first row of matrix



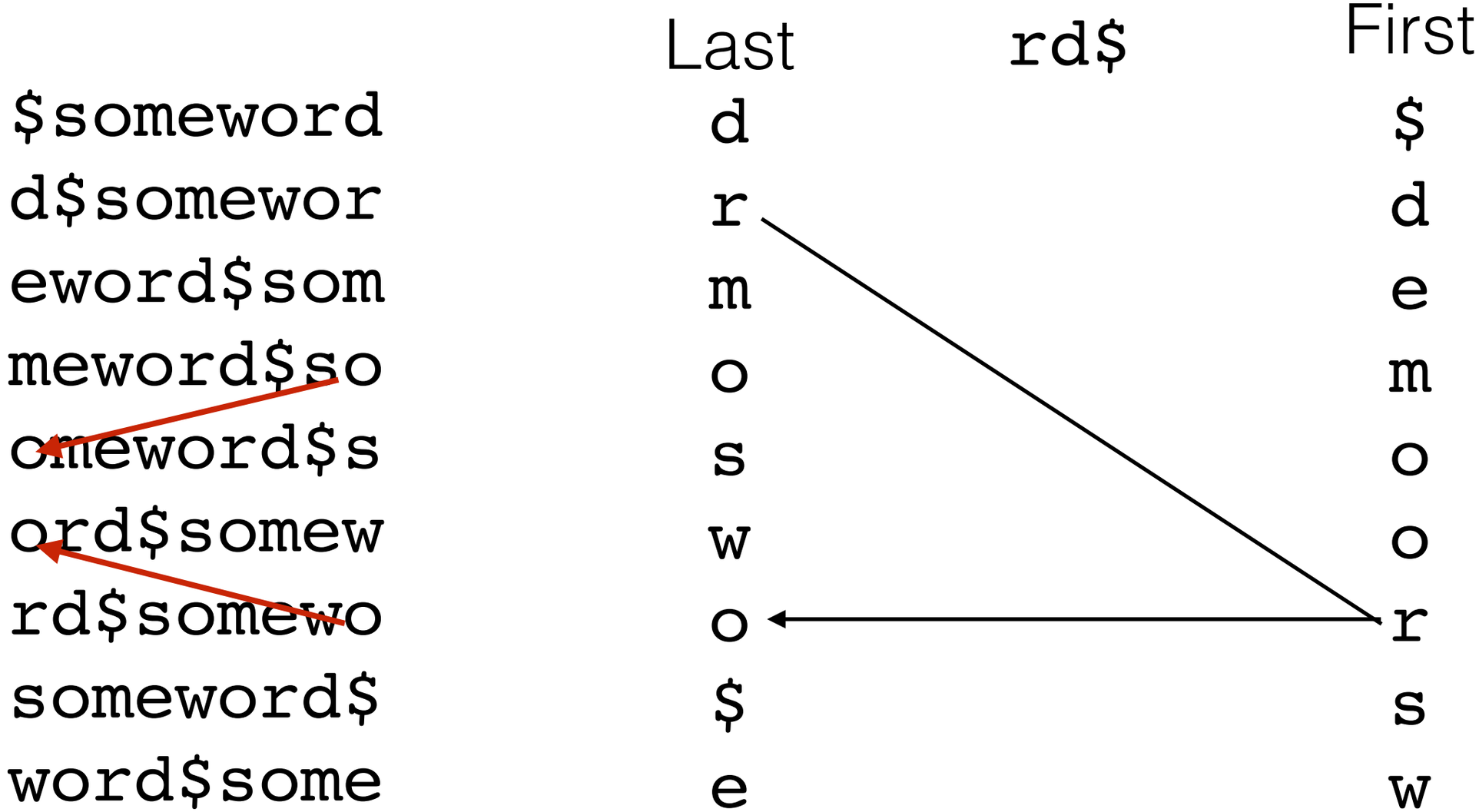
Last-First mapping makes BWT reversible

- The *i*th occurence of a letter in the BWT is same as *i*th occurence in first row of matrix



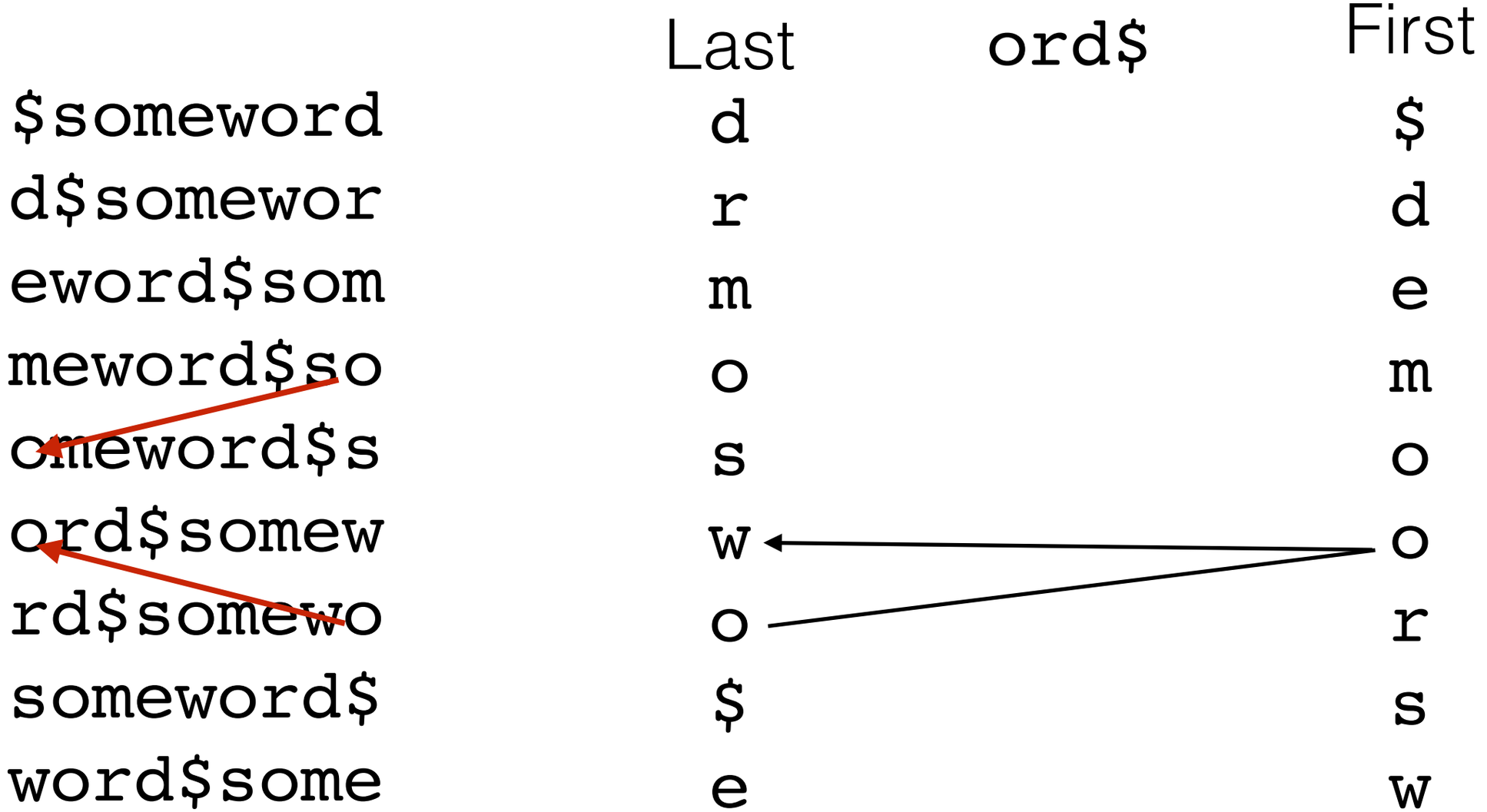
Last-First mapping makes BWT reversible

- The *i*th occurence of a letter in the BWT is same as *i*th occurence in first row of matrix



Last-First mapping makes BWT reversible

- The *i*th occurence of a letter in the BWT is same as *i*th occurence in first row of matrix



Last-First mapping makes BWT reversible

- The *i*th occurence of a letter in the BWT is same as *i*th occurence in first row of matrix

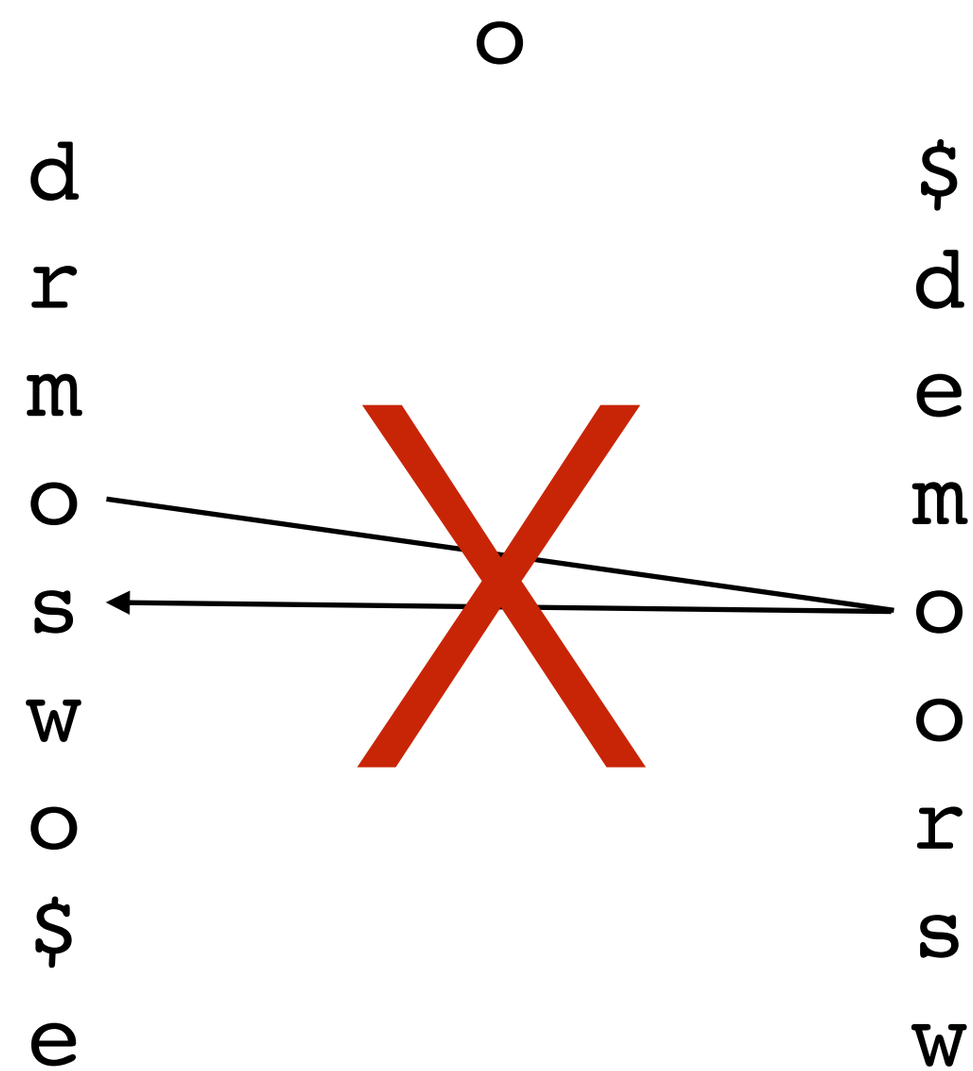
	Last		First
		`word$`	
`$someword`	`d`		`$`
`d$somewor`	`r`		`d`
`eword$som`	`m`		`e`
`mewords$so`	`o`	And so on…	`m`
`omeword$s`	`s`		`o`
`ord$somew`	`w`		`o`
`rd$somewo`	`o`		`r`
`somewords$`	`$`		`s`
`word$some`	`e`		`w`

This makes finding substrings easy

- Just start with a different character other than $!

Want to find

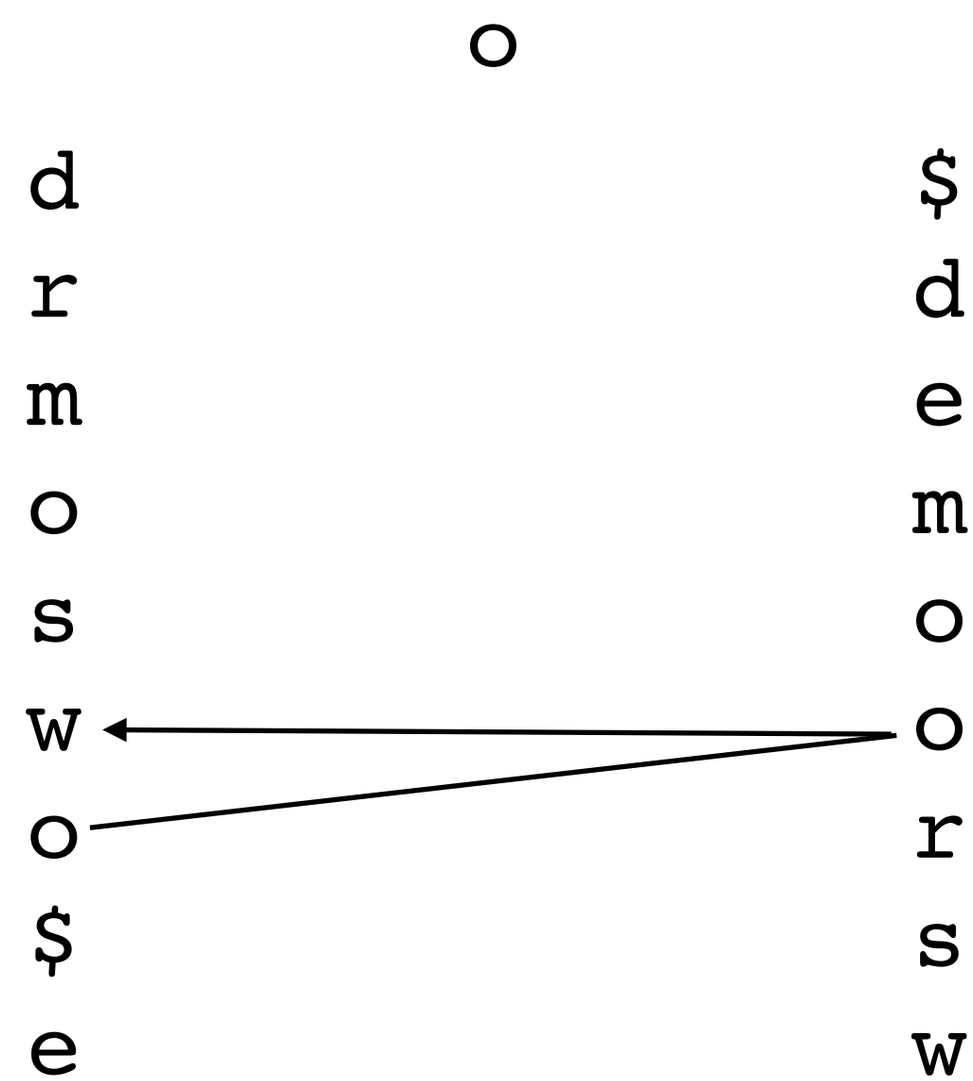
ewo



This makes finding substrings easy

- Just start with a different character other than $!

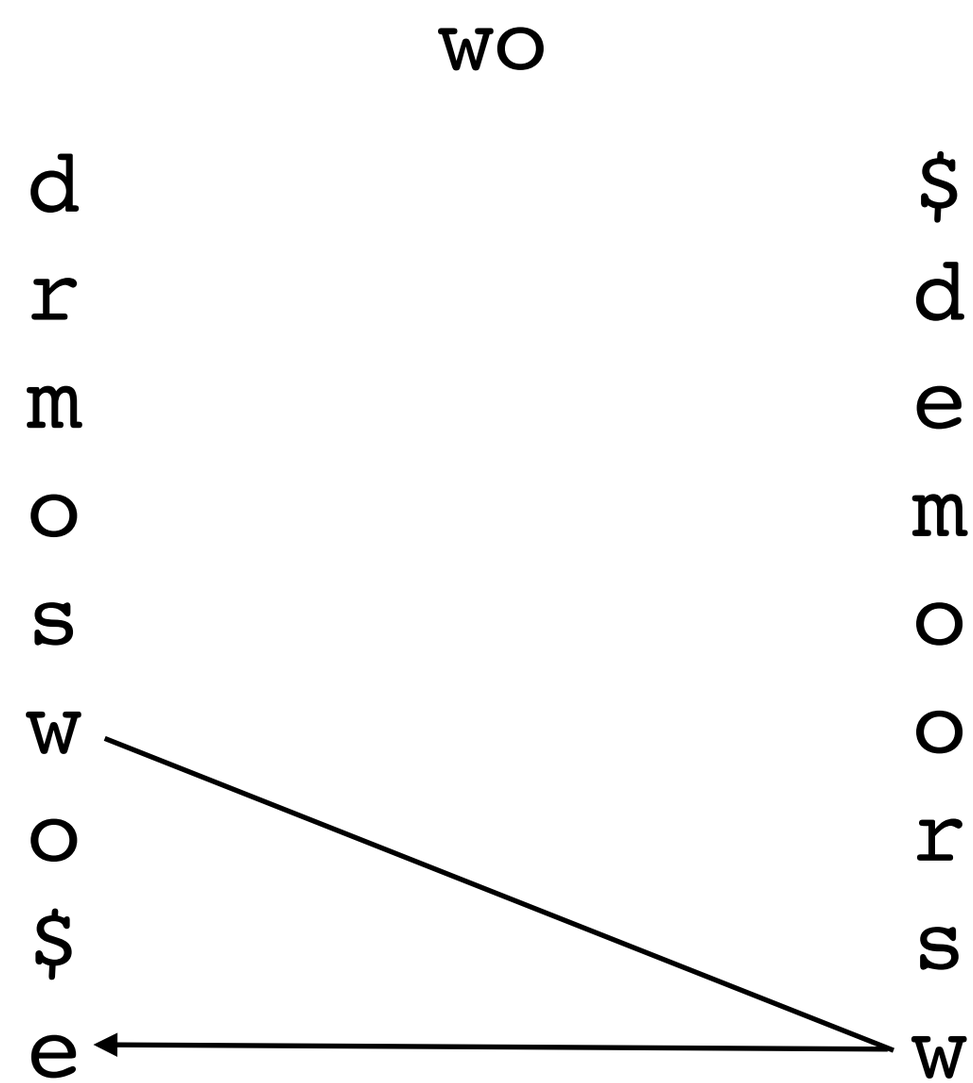
Want to find
ewo



This makes finding substrings easy

- Just start with a different character other than $!

Want to find
ewo



This makes finding substrings easy

- Just start with a different character other than $!

Want to find

ewo

ewo

d	$
r	d
m	e
o	m
s	o
w	o
o	r
$	s
e	w

Last-first mapping using two auxiliary tables

- A table C[c] that has the **C**ounts of every character lexicographically less than some character **c** in the string
    - This is small, only need one entry per possible character (so for DNA, only need A, C, G, T)
- A function Occ(c,k) that returns the number of **occ**urrences of character **c** up to position **k** in the BWT
    - We need to be clever about it.
- Then the position of Last[i] in First is C[Last[i]] + Occ(Last[i],i)
    - First is sorted, so all a character c will be at least at position C[c]
    - Then you need to know what occurrence of that character you're at in the BWT
    - O(1) time! So searching for a substring of length m is just O(m) time!

Some trips to help with the Occ(c,k) function

- Would be slow to compute every time, because would need to search the entire BWT!
- We want to precompute it
  - But precomputing will take a lot of space!
  - O(a*n) for an alphabet of a characters and string of length n
- Solution: precompute only for some k (say, every 10th)
  - Then backtrack a bit to add additional occurrences to your current point
  - Reduce storage space arbitrarily at the cost of some extra computing

Example of checkpointing Occ(c,k)

BWT	Occ(c,k)
$ a c a a c g	A:0 C:0 G:1 T:0
a a c g $ a c	
a c a a c g $	
a c g $ a c a	A:1 C:1 G:1 T:0
c a a c g $ a	
c g $ a c a a	
g $ a c a a c	A:3 C:2 G:1 T:0

NB: matrix is backwards from my examples

Image credit: Eleazar Eskin

Finding the position of a mapping

- Naively: “walk left” until you hit the beginning. Then just count how many steps you took!
  - This is a terrible idea
- Memory hungry: simply store the original position of every character in the BWT as an additional *suffix array*
  - Just read off the position of the first position in the read
  - Takes up an additional O(n) of space
- Compromise: Checkpoint and walk left

Checkpointing positions illustrated

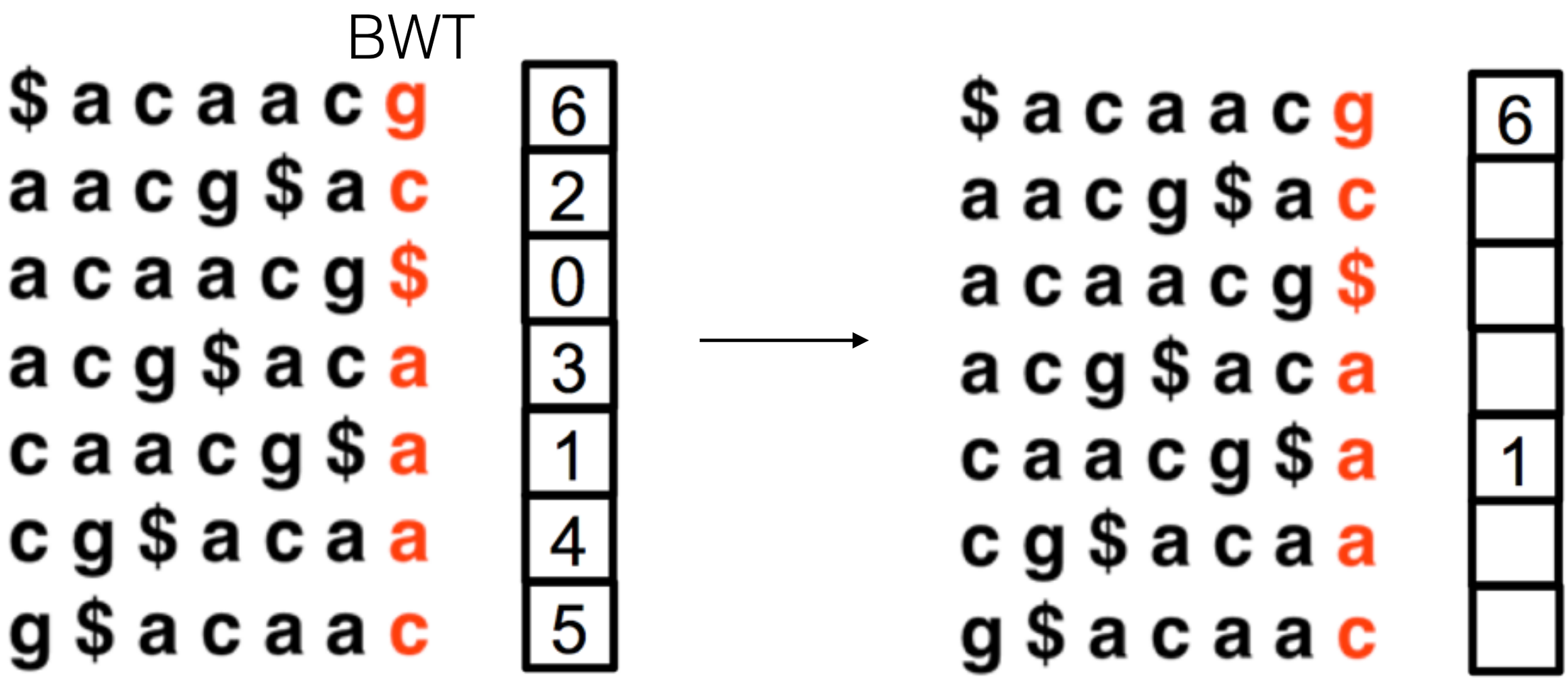


Image credit: Eleazar Eskin

Checking back in: why is all this useful

- Problem: genomes are big and we need to search a lot of reads
- Potential solution 1: hash the genome
  - Biggest strength: extremely fast
  - Biggest weakness: takes up a lot of memory
- Potential solution 2: index the genome
  - Biggest strength: very memory efficient
  - Biggest weakness: can be a lot slower than hashing

Using Bowtie 2 to actually map reads

- Bowtie was one of the first software to do read mapping using the BWT
  - Bowtie sort of sounds like “Bee Double-Yew Tee”
- Uses backtracing to deal with mismatches
  - Not going to cover in depth
  - Basic idea: if you don’t find a match, go back one position and try the other nucleotides and see if you now get a match
- Bowtie 2 efficiently handles gaps in the alignment
  - Essential for dealing with small indels

Bowtie 2 workflow

- Create FM-index out of reference genome
  - bowtie2-build
- Map reads
  - Paired end reads
  - Output unmapped reads
  - Sam file output

Bowtie-build

- bowtie-build /path/to/reference.fa /path/to/index
  - Creates a series of files, index.bt2.1, index.bt2.2, …

Bowtie2

- bowtie2 -x /path/to/index --un /path/to/unmapped.fastq -1 /path/to/reads_1.fastq -2 /path/to/reads_2.fastq -S /path/to/output.sam
  - index should be JUST the prefix, i.e. without the .bt2.1 parts
  - will output the unmapped reads to whatever is after --un
  - -S tells it to output a sam file

sam file format is the standard for mapped reads

- Has *all* the information that the fastq files have
  - All reads, mapped and unmapped
  - Read name
  - Read sequence
- Includes mapping information
  - Chromosome
  - Position
  - Strand (i.e. does it map forward or reverse?)
- Don’t worry about parsing sam files directly

bam files are *b*inary s*am* files

- Compressed and easily indexed
- Need to be converted from sam files

samtools is a software package for dealing with sam/bam files

- samtools view -bS input.sam > output.bam
  - Converts sam file into bam file. NB: the “>” character says to put the output in the file called output.bam
- samtools sort output.bam -o output.sorted.bam
  - Sorts the bam file into the file output.sorted.bam
- samtools index output.sorted.bam
  - Will index the bam file so that it can be easily accessed
- NB: You will have all the intermediate files left over (e.g. the sam file, the unsorted bam file). You can delete them!

pysam is a powerful Python API for dealing with bam files

- bamfiles can be opened using the AlignmentFile interface
  - bamfile = pysam.AlignmentFile(“filename.bam”, “rb”)
- You can iterate through bamfiles with fetch
  - for read in bamfile.fetch():
  - can fetch specific regions: bamfile.fetch(“chrom”,start,stop)
- reads have specific atributes
  - Look up the readthedocs!