Computational Genetics
Spring 2014
Lecture 6

Eleazar Eskin

University of California, Los Angeles

Home Work & Midterm

- HW1 Due 4/17/13
- HW2 Due 4/22/13
- Power Tagging Paper Question due Monday (4/21/14)
- Power Tagging Paper Responses due Wednesday (4/23/14)
- Project Selection was due (4/11/14)

- Midterm Review (4/21/14)
- Midterm (4/23/14)

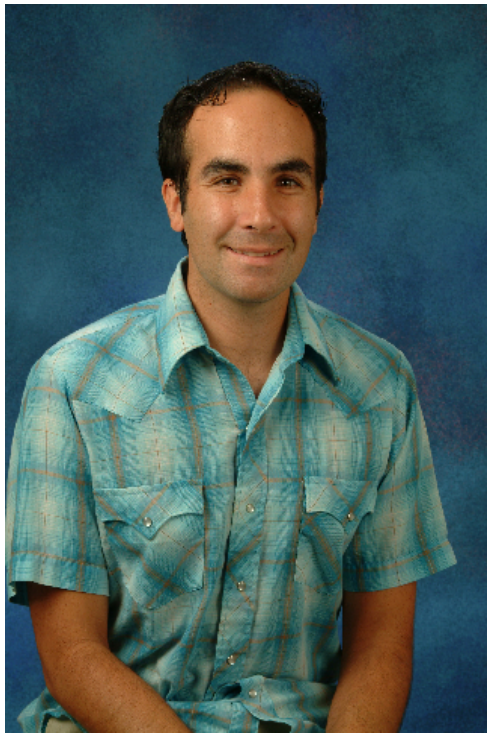
“Re”-Sequencing + Burroughs Wheeler Transform

Lecture 6.

April 16th, 2014

(Some slides from Ben Langmead)

How do we get someone's DNA sequence? Where are my mutations?



Sequencing Technology

Illumina / Solexa
Genetic Analyzer 1G
1000 Mb/run, 35bp reads

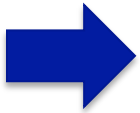
AGAGCAGTCGAC
AGGTATAGTCTA
CATGAGATCGAC
ATGAGATCGGTA
GAGCCGTGAGAT
CGACATGATAGC
CAGAGCAGTCGA
CAGGTATAGTCT
ACATGAGATCGA
CATGAGATCGGT
AGAGCCGTGAGA
TCGACATGATAG
CCAGAGCAGTCG
ACAGGTATAGTC
TACATGAGATCG
ACATGAGATCGG
TAGAGCCGTGAG
ATCGACATGATA
GCCAGAGCAGTC
GACAGGTATAGT
CTACATGAGATC
GACATGAGATCG
GTAGAGCCGTGA
GATCGACATGAT

- Next generation sequencing.
  - **Cheap sequencing.**
  - **"Short Reads"**

Short Read Sequencing Problem (A Computer Science Problem)

Full DNA Sequence

AGAGCAGTCGAC
AGGTATAGTCTA
CATGAGATCGAC
ATGAGATCGGTA
GAGCCGTGAGAT
CGACATGATAGC
CAGAGCAGTCGA
CAGGTATAGTCT
ACATGAGATCGA
CATGAGATCGGT
AGAGCCGTGAGA
TCGACATGATAG
CCAGAGCAGTCG
ACAGGTATAGTC
TACATGAGATCG
ACATGAGATCGG
TAGAGCCGTGAG
ATCGACATGATA
GCCAGAGCAGTC
GACAGGTATAGT
CTACATGAGATC



- Short read sequencers generate random short substrings from the DNA sequence of a certain length.

ATGAGATCGGTAGAGCCGTGAGAT
GAGCAGTCGACAGGTATAGTCTAC
AGAGCAGTCGACAGGTATAGTCTA
TGAGATCGACATGATAGCCAGAGC
TAGCCAGAGCAGTCGACAGGTATA
GATAGCCAGAGCAGTCGACAGGTA
GAGATCGACATGATAGCCAGAGCA
GCAGTCGACAGGTATAGTCTACAT
AGCAGTCGACAGGTATAGTCTACA
TCGACATGAGATCGGTAGAGCCGT
CAGTCGACAGGTATAGTCTACATG
GAGATCGACATGATAGCCAGAGCA
GTAGAGCCGTGAGATCGACATGAT

Short Reads Difficulties

```
ATGAGATCGGTAGAGCCGTGAGAT
GAGCAGTCGACAGGTATAGTCTAC
AGAGCAGTCGACAGGTATAGTCTA
TGAGATCGACATGATAGCCAGAGC
TAGCCAGAGCAGTCGACAGGTATA
GATAGCCAGAGCAGTCGACAGGTA
GAGATCGACATGATAGCCAGAGCA
GCAGTCGACAGGTATAGTCTACAT
AGCAGTCGACAGGTATAGTCTACA
TCGACATGAGATCGGTAGAGCCGT
CAGTCGACAGGTATAGTCTACATG
GAGATCGACATGATAGCCAGAGCA
GTAGAGCCGTGAGATCGACATGAT
```

- We don't know where each read comes from!
- Can't identify where the mutations are!
- What do we do?

Key Idea: "Re"-Sequencing

We know that my genome is very close to the Human genome.

My Genome:

TACATGAGATCGACATGAGATCGGTAGAGCCGTGAGATC

A Sequence Read:

TCGACATGAGATCGGTAGAGCCGT

The Human Genome:

TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC
TCGACATGAGATCGGTAGAGCCGT

Recovered Sequence:

TACATGAGATCGACATGAGATCGGTAGAGCCGTGAGATC

"Re"-Sequencing Output

Resequencing provides a list of changes to make from the reference to change it to the target. Similar to unix "diff".

My Genome:

TACATGAGATC**G**ACATGAGATC**G**GTAGAGC**C**GTGAGATC

The Human Genome:

TACATGAGATC**G**ACATGAGATC**G**GTAGAGC**C**GTGAGATC

A list of changes

Recovered Sequence:

TACATGAGATC**G**ACATGAGATC**G**GTAGAGC**C**GTGAGATC

Algorithmic “Re”-Sequencing Challenges

- Sequences are long!
  - **Human Genome is 3,000,000,000 long.**
- Sequencers generate many reads!
  - **A sequencer generates over 1,000,000,000 reads.**

- We need efficient algorithms to “map” each read to its location in the genome.

There are other challenges which we are not mentioning.

Trivial Mapping Algorithm

The Human Genome:
TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC

A Sequence Read:
TCGACATGAGATCGGTAGAGCCGT

- We can slide our read along the genome and count the total mismatches between the read and the genome.
- If the mismatches are below a threshold, we say that it is a match.

TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC
TCGACATGAGATCGGTAGAGCCGT

Total of 18 mismatches. Not below threshold. Not a match.

Trivial Mapping Algorithm

The Human Genome:

TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC

A Sequence Read:

TCGACATGAGATCGGTAGAGCCGT

TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC
TCGACATGAGATCGGTAGAGCCGT

Total of 15 mismatches. Not below threshold. Not a match.

Trivial Mapping Algorithm

The Human Genome:
TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC

A Sequence Read:
TCGACATGAGATCGGTAGAGCCGT

TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC
  TCGACATGAGATCGGTAGAGCCGT

Total of 23 mismatches. Not below threshold. Not a match.

Trivial Mapping Algorithm

The Human Genome:
TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC

A Sequence Read:
TCGACATGAGATCGGTAGAGCCGT

TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC
TCGACATGAGATCGGTAGAGCCGT

Total of 23 mismatches. Not below threshold. Not a match.

Trivial Mapping Algorithm

The Human Genome:

```
TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC
```

A Sequence Read:

```
TCGACATGAGATCGGTAGAGCCGT
```

```
TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC
          TCGACATGAGATCGGTAGAGCCGT
```

Total of 3 mismatches. Below threshold. A match!

Complexity of Trivial Algorithm

- 3,000,000,000 length genome (N)
- 300,000,000 reads to map (M)
- Reads are of length 30 (L)
- Number of mismatches allowed is 2 (D).
- Each comparison of match vs. mismatch takes 1/1,000,000 seconds (t).

Total Time = N*M*L*t = 27,000,000,000,000 seconds or 864,164 years!

- Important: Trivial algorithm only solves problem under assumptions.

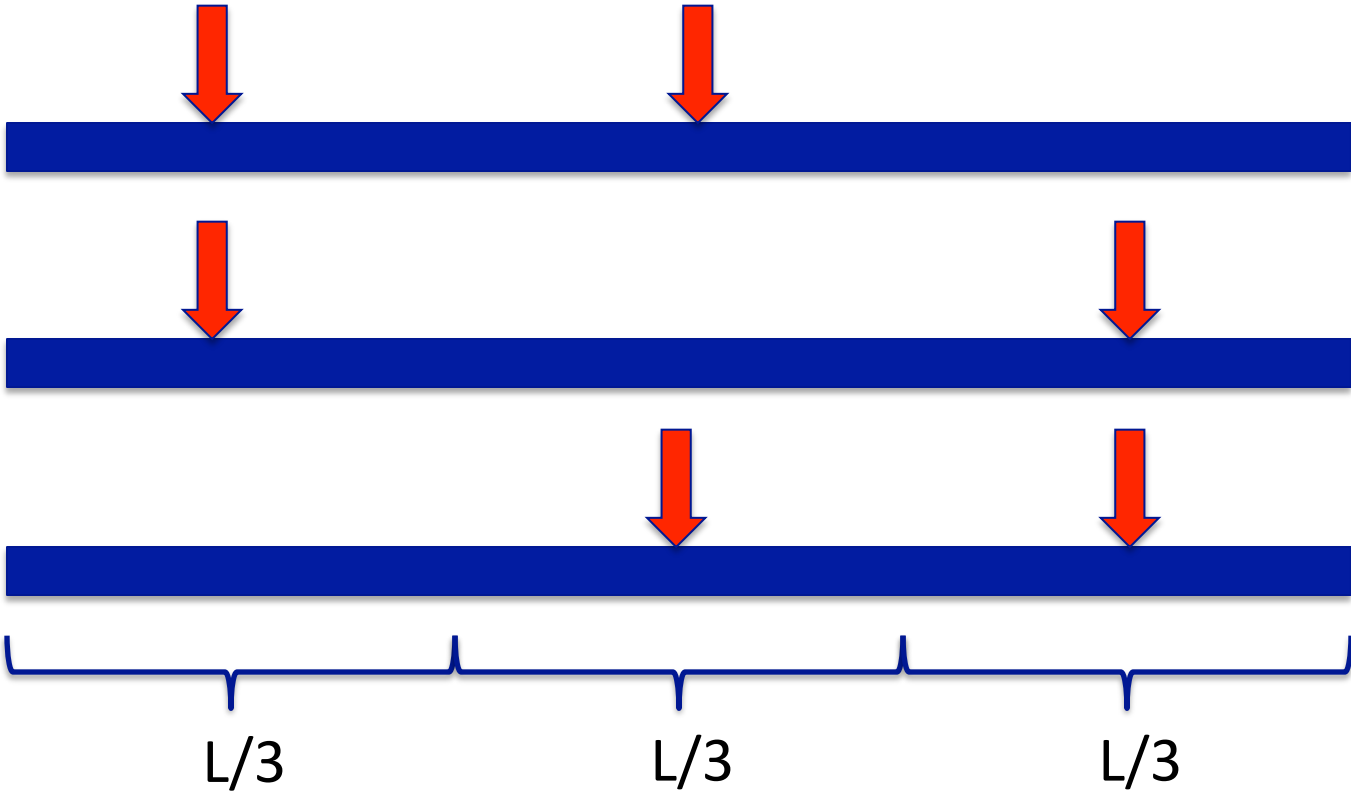
Some observations

- Most positions in the genome match very poorly.
- We are looking for only a few mismatches.

  (D is small)
- A substring of our read will match perfectly.

Perfect Matching Read Substrings

Three "worst" possible cases for placement of mutations.



- In each case, there is a perfect match of L/3.

Finding a perfect match of length L/3

- 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.

Complexity of Indexing Algorithm

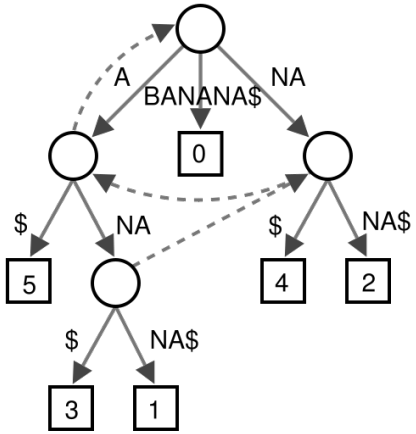
- We need to look up each third of the read in the index.
- For L=30, our index will contain entries of length 10. Each entry will contain on average $N/(4^{L/3})$ or 3,000 positions.
- For each position, we need to compute the number of mismatches.
- Our running time is $L * M*3*N/(4^{L/3})*T$=81,000,000 seconds or 937 days.
- If L=45, then the time is 81,000 seconds or 22.5 hours.

Indexing a genome

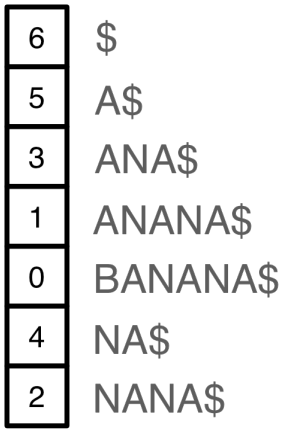
- To find exactly matching substrings, we need to build an index for the whole genome.
- Problem: The genome is BIG!

Indexing

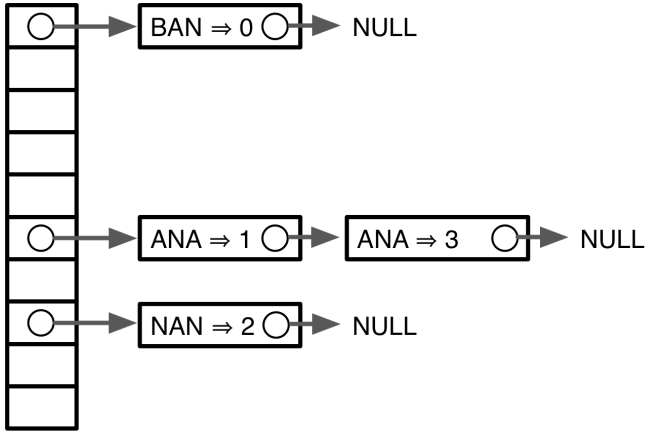
- Genome indices can be big. For human:



> 35 GBs



> 12 GBs

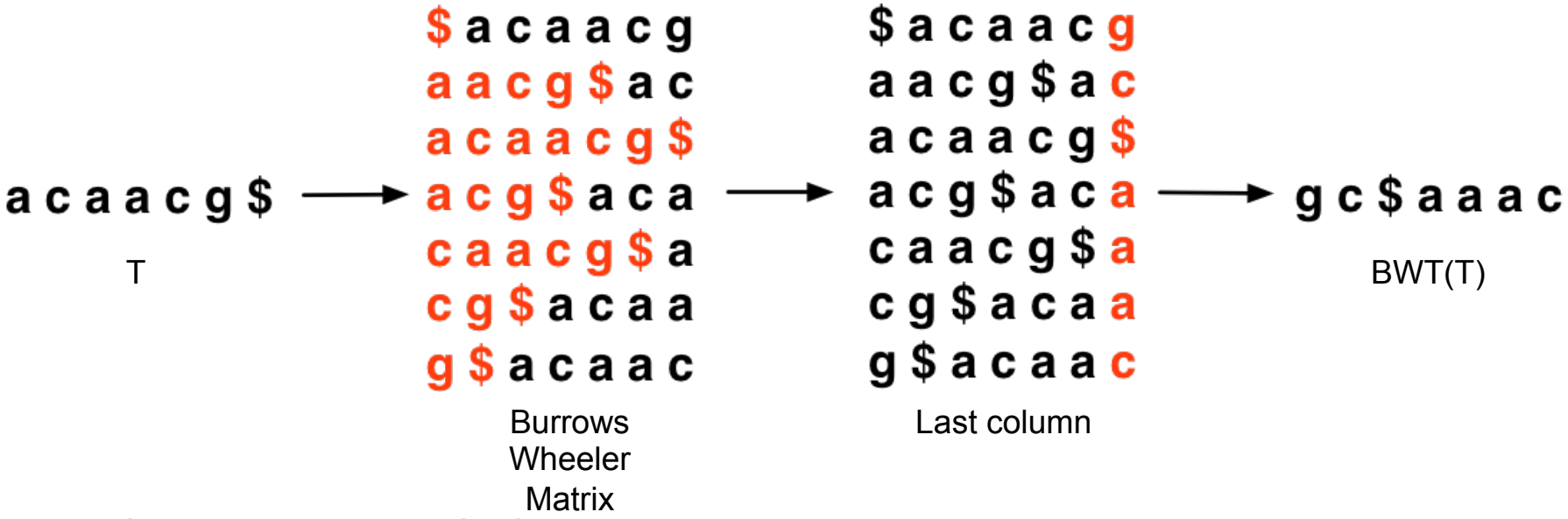


> 12 GBs

- Large memory requirement implications
  - **Requires large memory machine (expensive)**
  - **Partition genome and index each part (slow)**

Burrows-Wheeler Transform

- http://en.wikipedia.org/wiki/Burrows-Wheeler_transform
- Reversible permutation used originally in compression

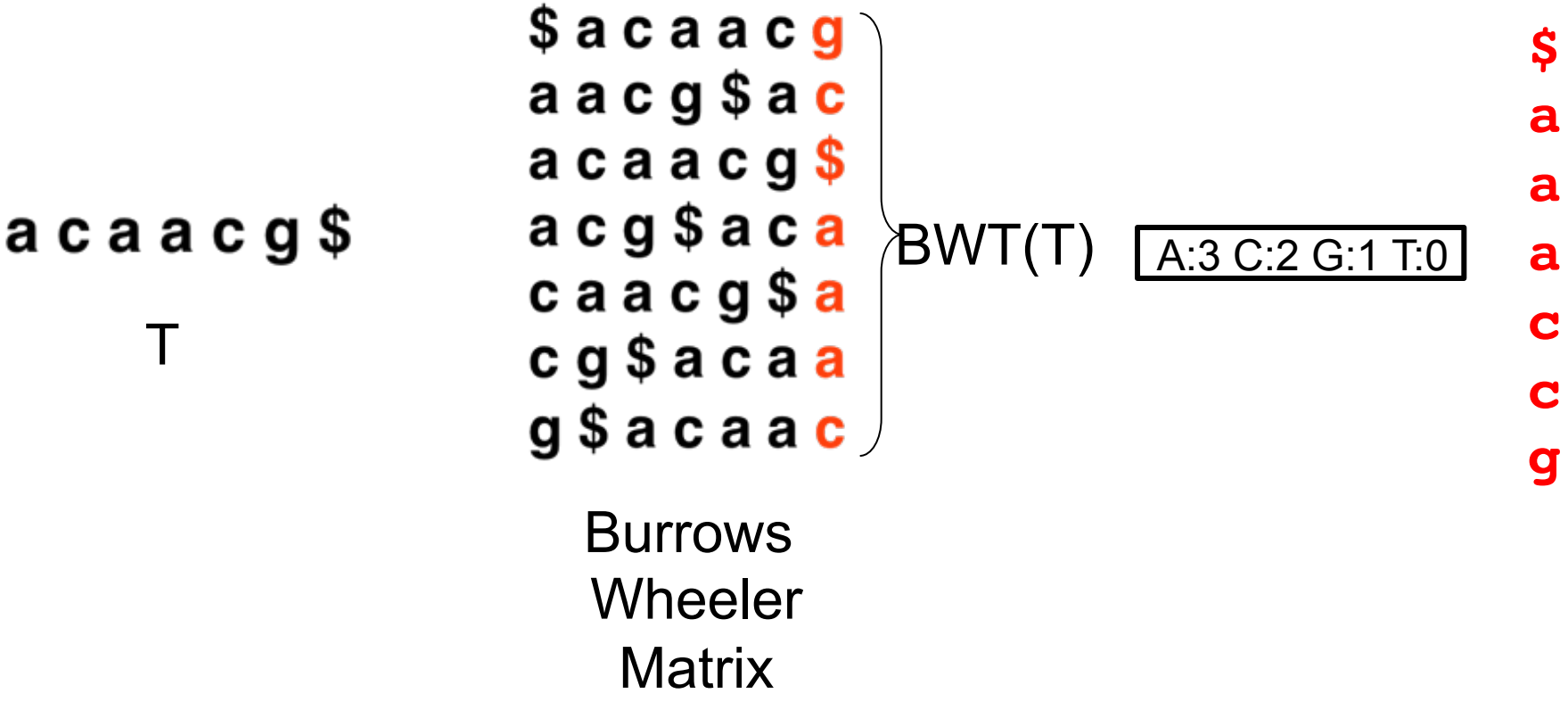


- Once BWT(T) is built, *all else shown here is discarded*
  - **Matrix will be shown for illustration only**

Burrows M, Wheeler DJ: A block sorting lossless data compression algorithm. *Digital Equipment Corporation, Palo Alto, CA* 1994, Technical Report 124; 1994

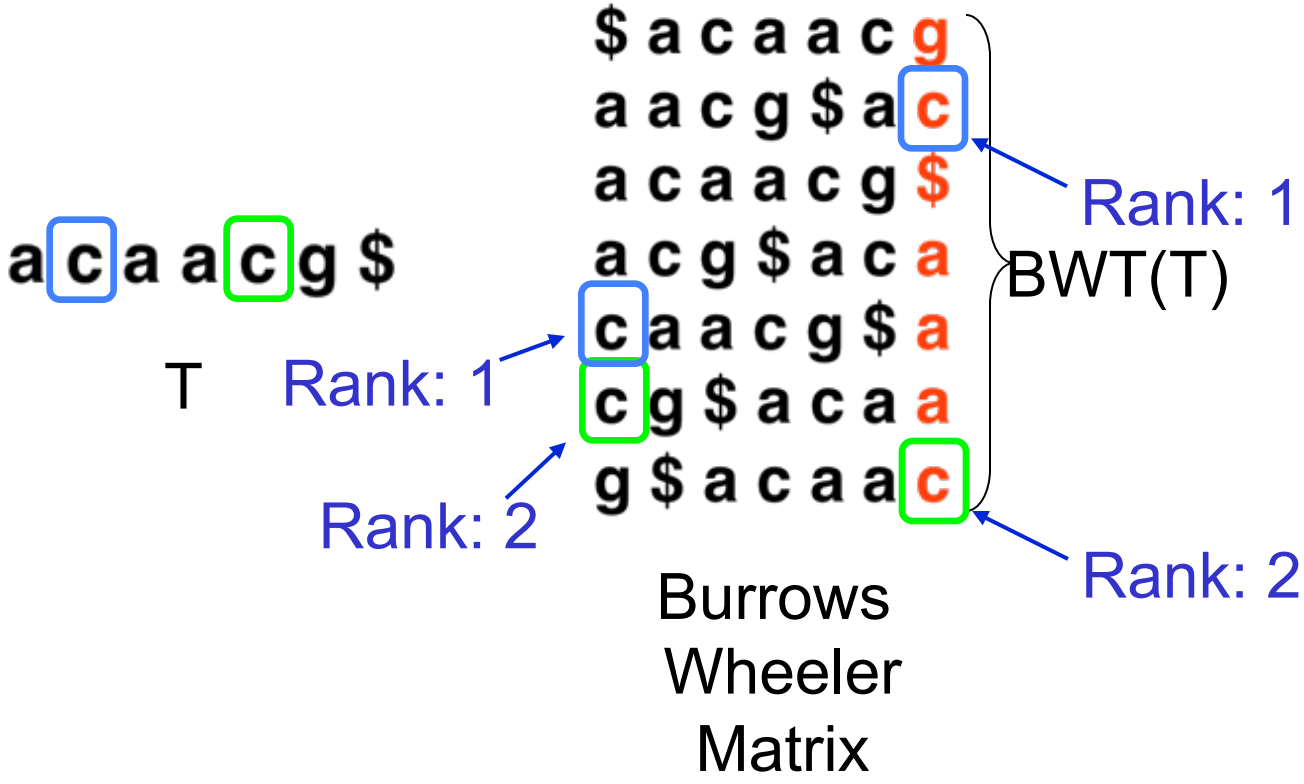
Burrows-Wheeler Transform

- Store only last column
- First column can be recovered by counting symbols in last column because it is sorted



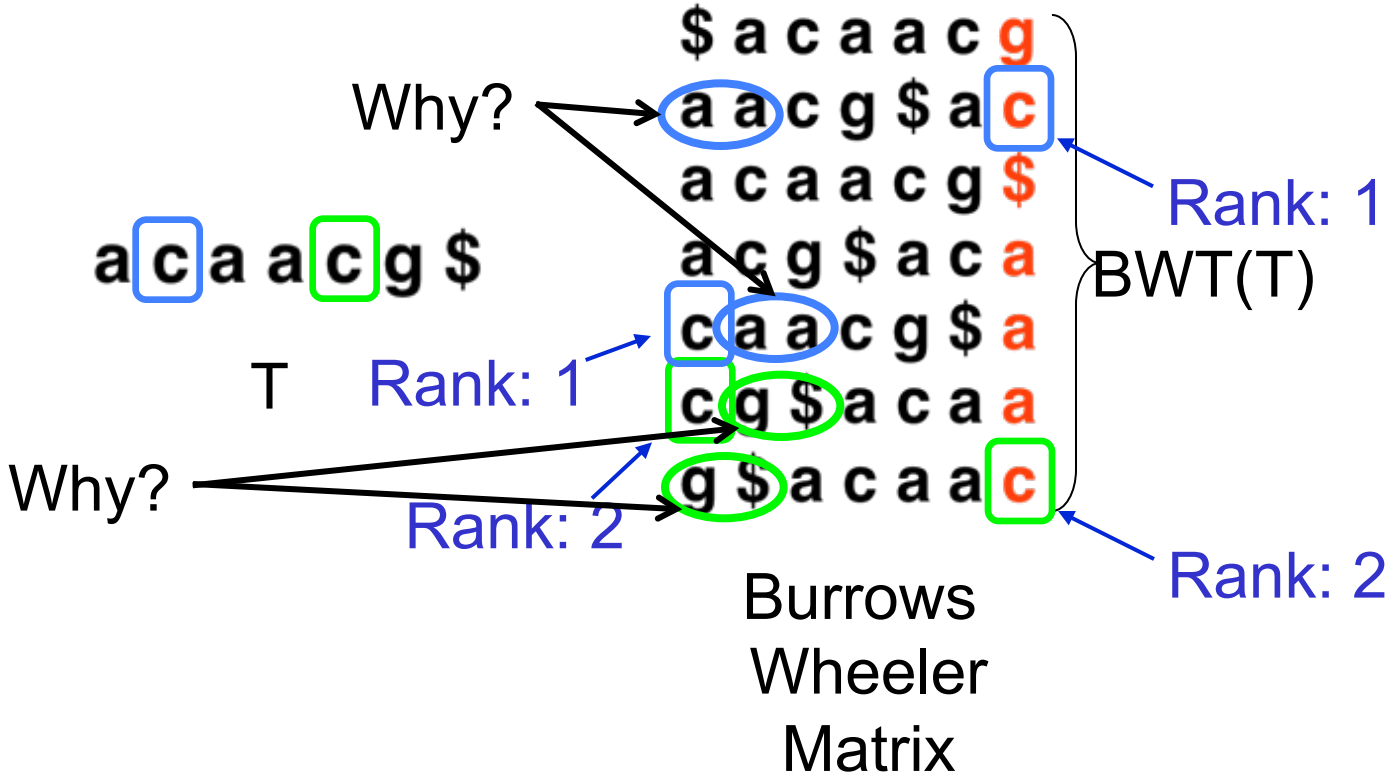
Burrows-Wheeler Transform

- Property that makes BWT(T) reversible is "LF Mapping"
  - **i^{th} occurrence of a character in Last column is same *text* occurrence as the i^{th} occurrence in First column**



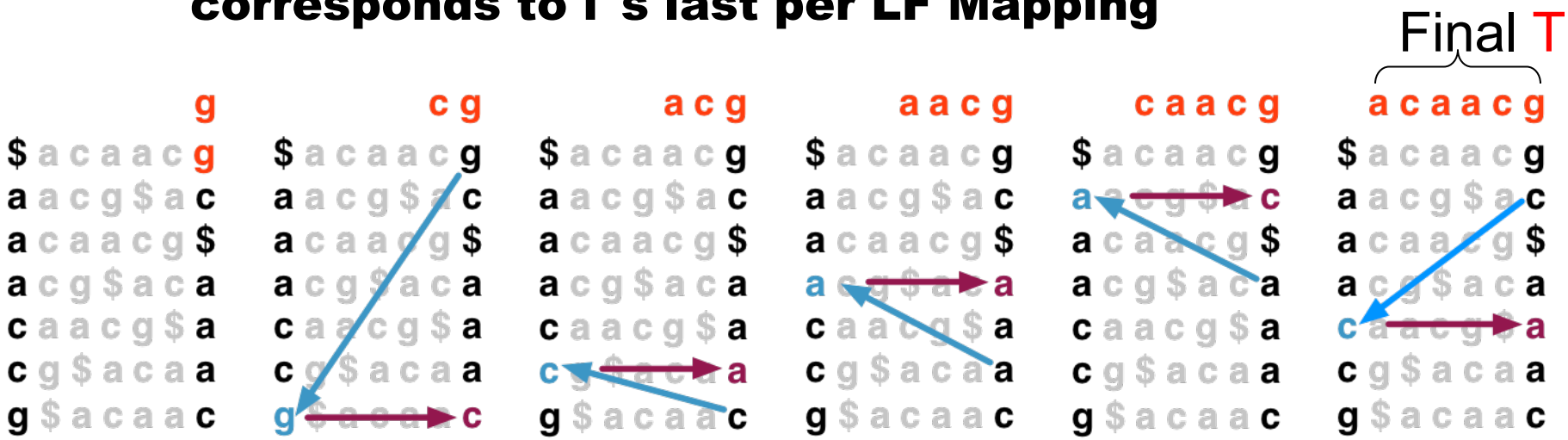
Burrows-Wheeler Transform

- Property that makes BWT(T) reversible is "LF Mapping"
  - **i^{th} occurrence of a character in Last column is same *text* occurrence as the i^{th} occurrence in First column**



Burrows-Wheeler Transform

- To recreate T from BWT(T), repeatedly apply rule:
  - **T = BWT[LF(i)] + T; i = LF(i)**
  - **Where LF(i) maps row i to row whose first character corresponds to i's last per LF Mapping**



- Could be called "unpermute" or "walk-left" algorithm

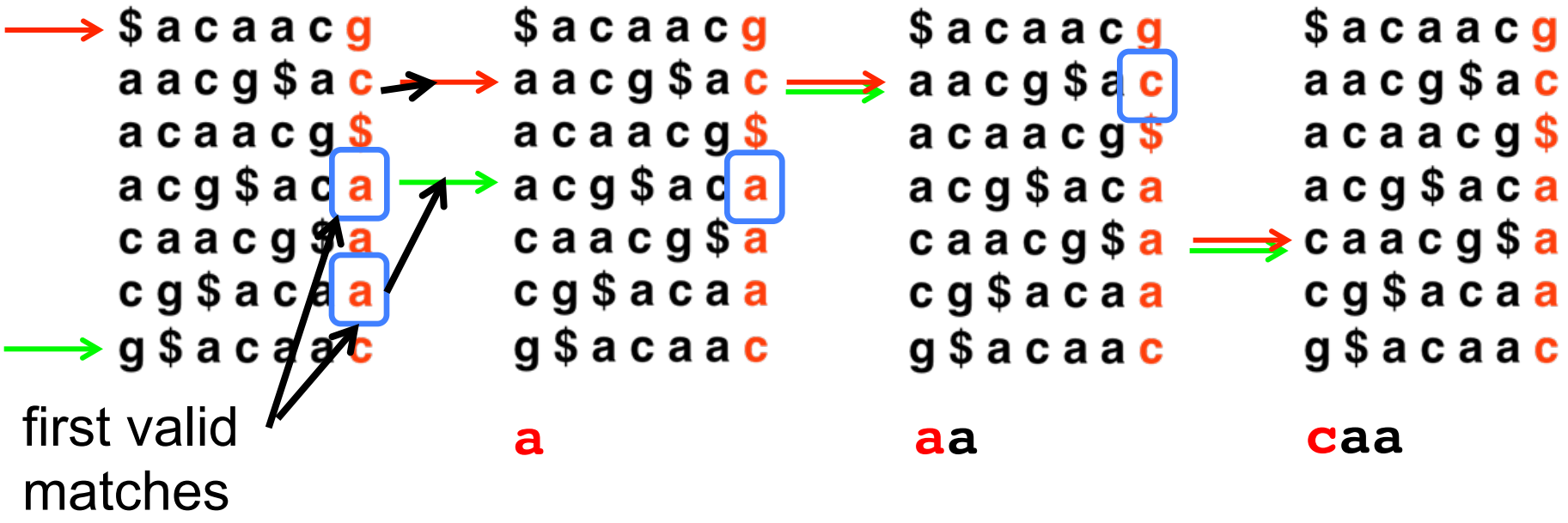
FM Index

- Ferragina & Manzini propose "FM Index" based on BWT
- Observed:
  - **LF Mapping also allows *exact matching* within T**
  - **LF(i) can be made fast with *checkpointing***
  - **...and more (see FOCS paper)**

- Ferragina P, Manzini G: Opportunistic data structures with applications. *FOCS. IEEE Computer Society; 2000.*
- Ferragina P, Manzini G: An experimental study of an opportunistic index. *SIAM symposium on Discrete algorithms*. Washington, D.C.; 2001.

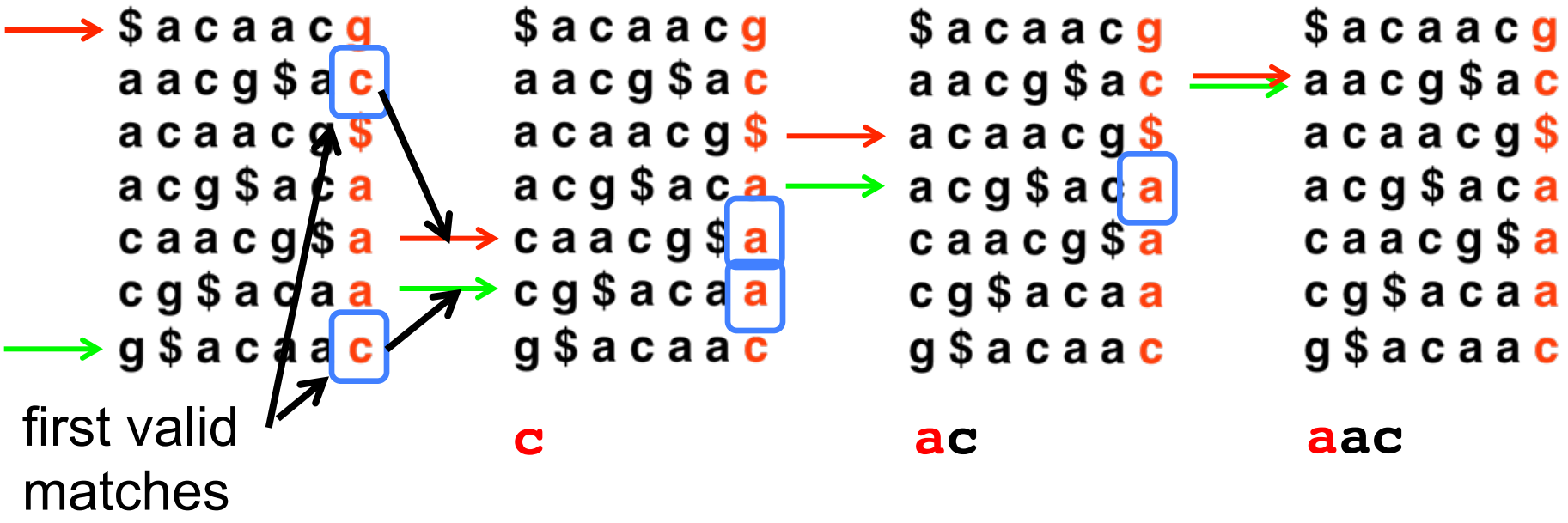
Exact Matching with FM Index

- Look up pattern in reverse.
- Use 2 pointers to represent range of matches.
  - **All matches will be next to each other in matrix.**
- Find first valid match for next symbol in range.
  - **Example: searching for "caa"**



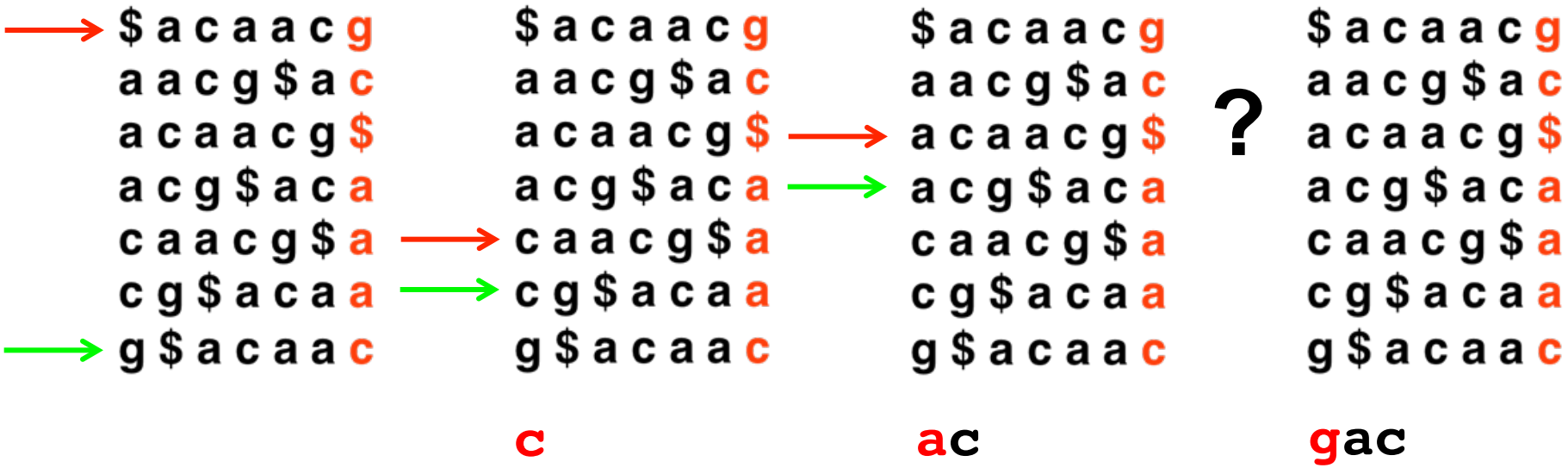
Exact Matching with FM Index

- Look up pattern in reverse.
- Use 2 pointers to represent range of matches.
- Find first valid match for next symbol in range.
  - **Example: searching for "aac"**



Exact Matching with FM Index

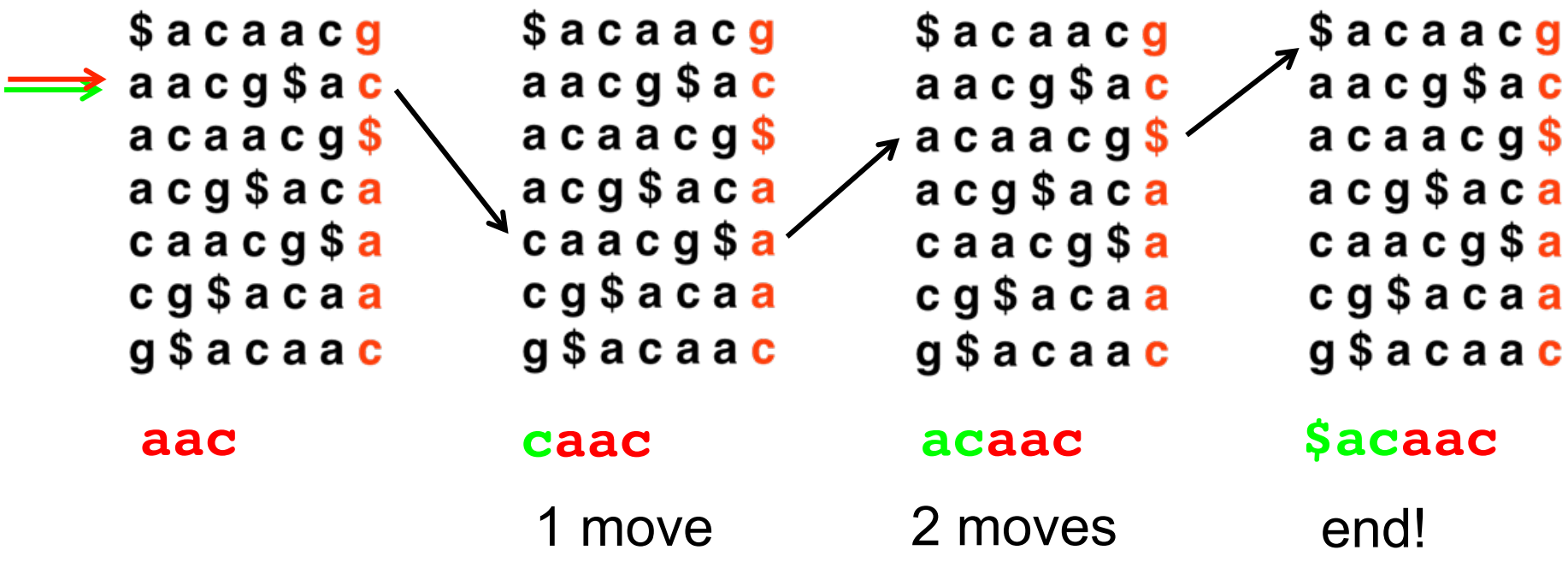
- If no match…
  - **Example: searching for "gac"**



- Pointers will get lost.
- FM index can quickly check for a match.

Where in sequence is the match?

- Use “walk-left” to build sequence to start
- Count number of sequences
  - **Example: searching for “aac”**



- Number of moves back is start position of match
  - **Example: “aac” is in position 2.**

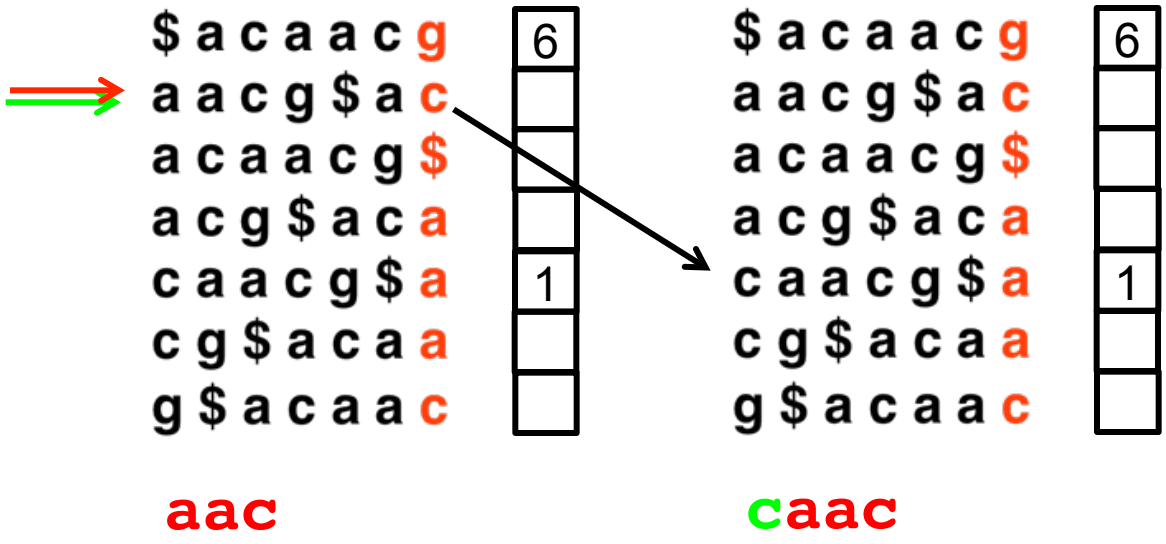
Where in sequence is match?

- "walk-left" to start of sequence is slow
- Requires on average N/2 steps to reach start.
- Alternate strategy: keep index of positions.
  - **Example: searching for "aac"**

- Problem: requires as much storage as hashtable!

Where in sequence is match?

- Key Idea: Store fraction of array (sampling)
- Only store some positions and "walk-left"
- Combines two previous strategies
  - **Example: searching for "aac"**



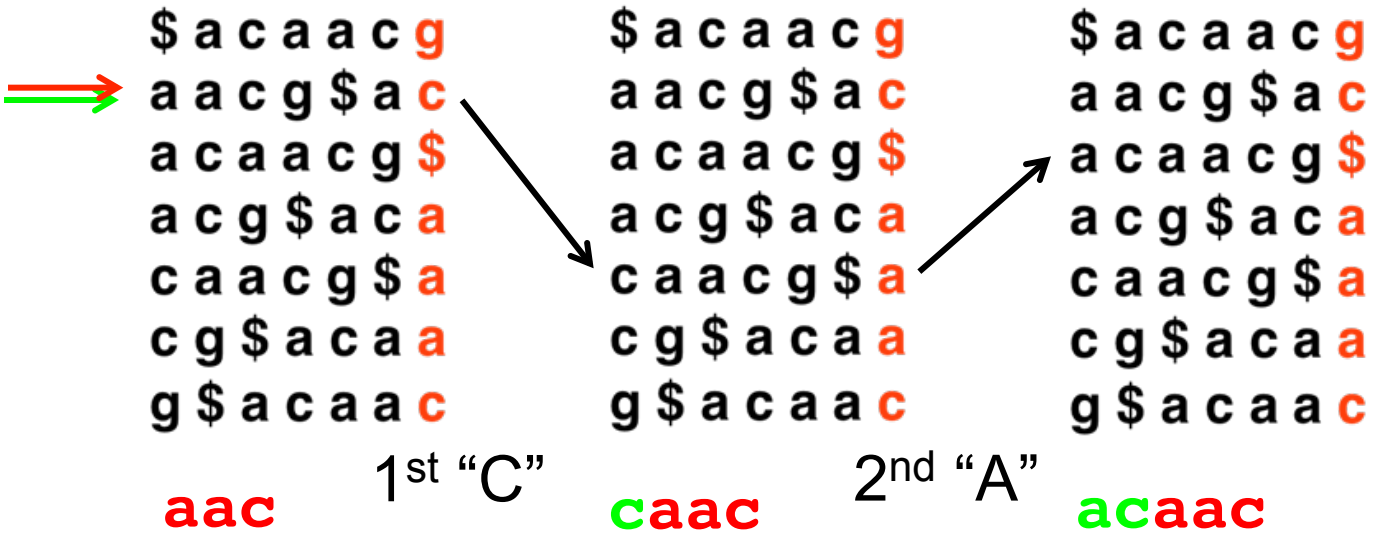
Position =
number of moves
+ position in array

For "aac" = 1 + 1 = 2

- How many values to store provides defines time/space tradeoff.

“walk-left” optimization

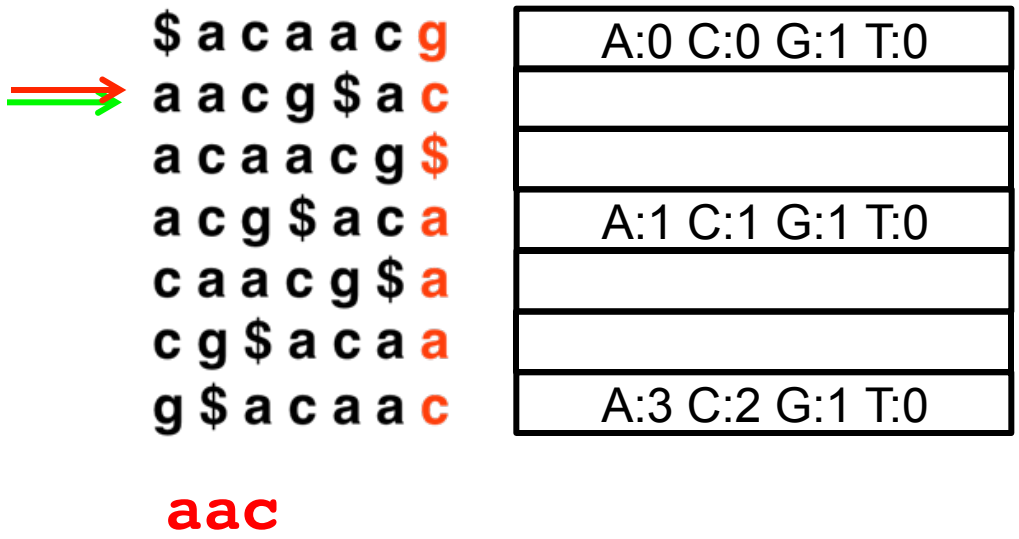
- Each “walk-left” requires counting previous occurrences of symbol in BWT
  - **Example: searching for “aac”**



- Requires counting occurrences in N/2 length string
- Really slow!

“walk-left” optimization

- Idea: use checkpoints to store previous counts
  - **Example: searching for “aac”**



- Requires counting occurrences only until checkpoint.
- Really fast!

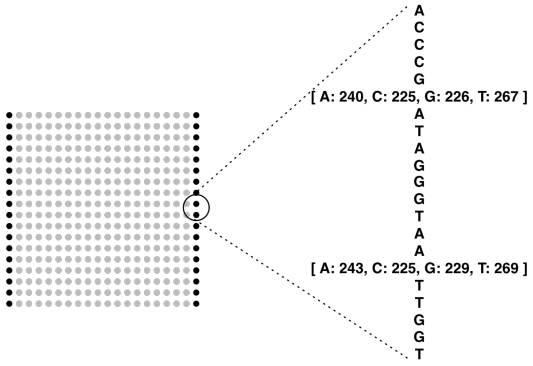
FM Index is Small (Bowtie)

- Entire FM Index on DNA reference consists of:
  - **BWT (same size as T)**
  - **Checkpoints (~15% size of T)**
  - **SA sample (~50% size of T)**
- Total: ~1.65x the size of T

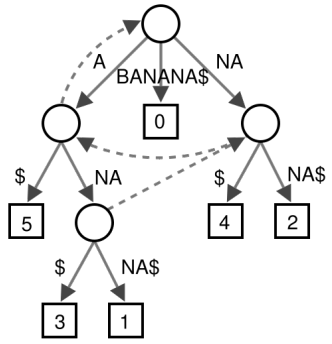
Assuming 2-bit-per-base encoding and no compression, as in Bowtie

Assuming a 16-byte checkpoint every 448 characters, as in Bowtie

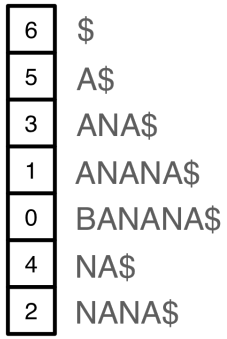
Assuming Bowtie defaults for suffix-array sampling rate, etc



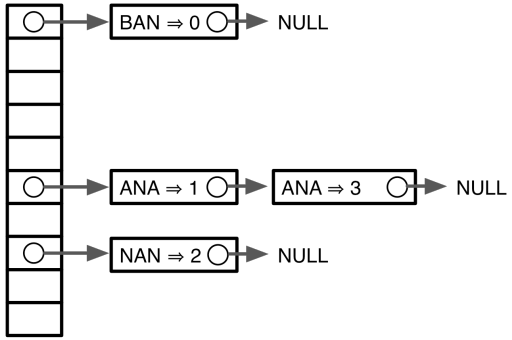
~1.65x



>45x



>15x



>15x

Reference Paper

- Langmead B, Trapnell C, Pop M, Salzberg SL. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biology 10:R25.
  - (Some slides from paper)