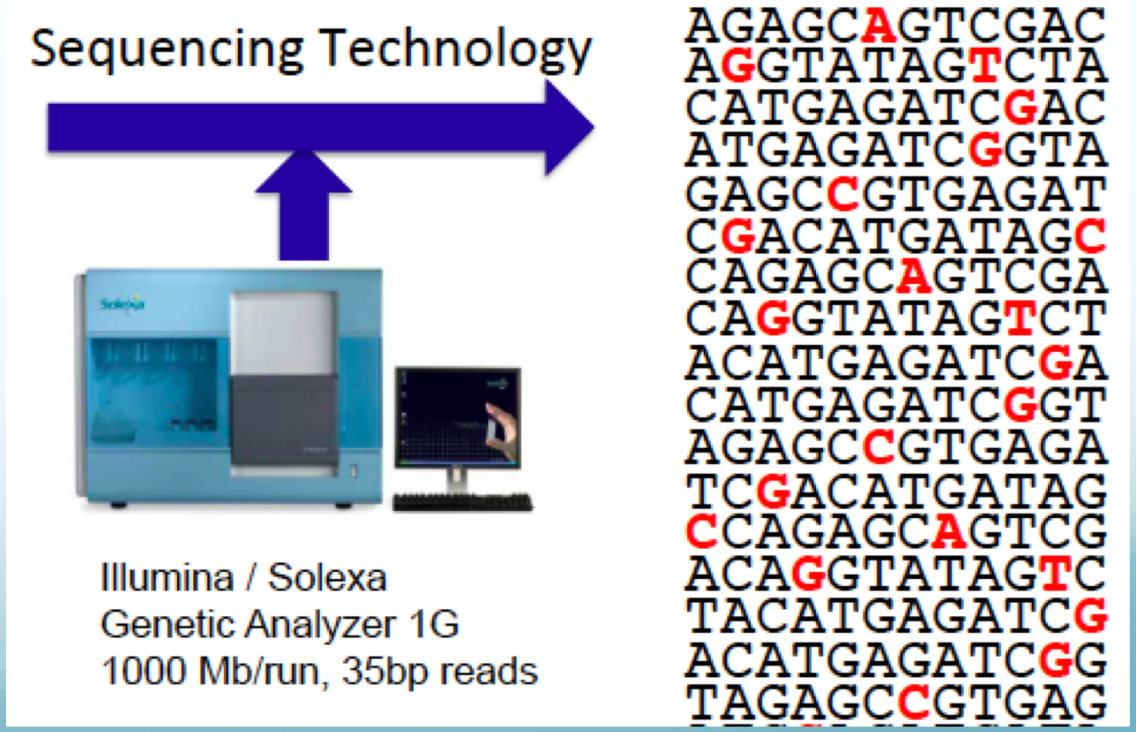
Ultrafast String matching algorithm

Lecture 10

perfect match

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

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

Reference Human Genome

- 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

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 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.

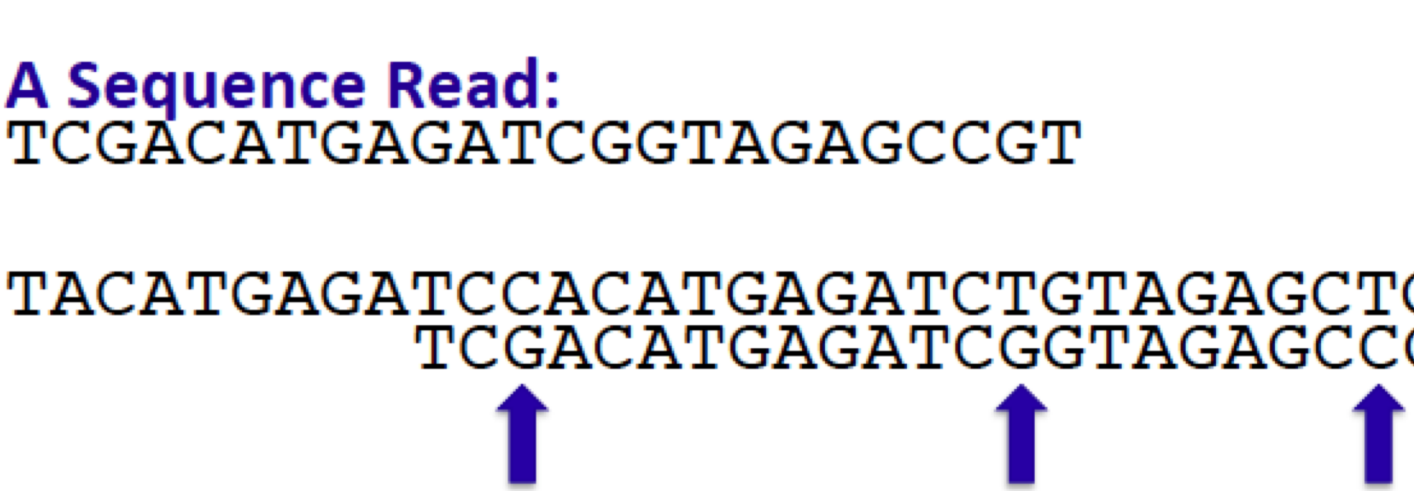
Trivial Mapping Algorithm

The Human Genome:

TACATGAGATCCACATGAGATCTGTAGAGCTGTGAGATC

A Sequence Read:

TCGACATGAGATCGGTAGAGCCGT



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).
- Important: Trivial algorithm only solves problem under assumptions.
- Total Time = ?

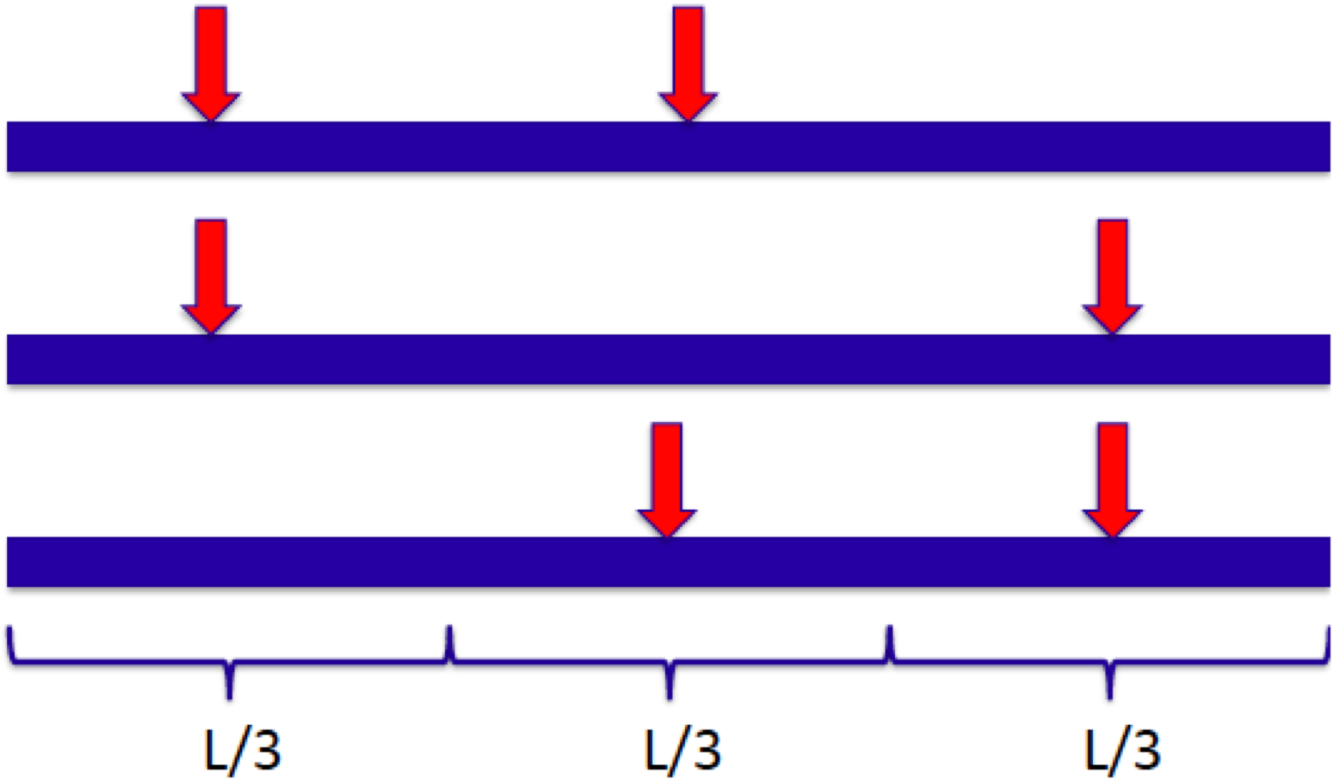
Some observations

- 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

- For the case allowing 2 mismatches for read of

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.

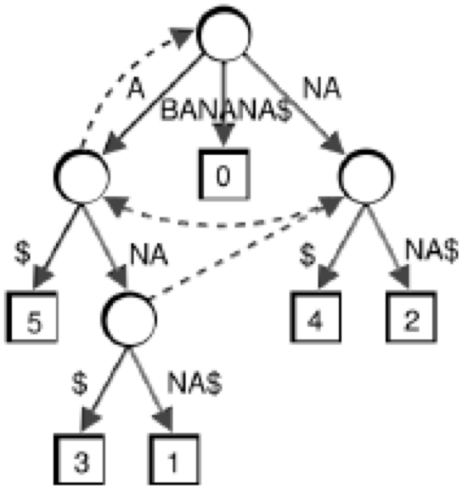
Sequence	Positions
AAAAAAAAAA	32453, 64543, 76335
AAAAAAAAAC	64534, 84323, 96536
AAAAAAAAAG	12352, 32534, 56346
AAAAAAAAAT	23245, 54333, 75464
AAAAAAAACA	
AAAAAAAACC	43523, 67543
…	
CAAAAAAAAA	32345, 65442
CAAAAAAAAC	34653, 67323, 76354
…	
TCGACATGAG	54234, 67344, 75423
TCGACATGAT	11213, 22323
…	
TTTTTTTTTG	64252
TTTTTTTTTT	64246, 77355, 78453

Complexity of Indexing Algorithm

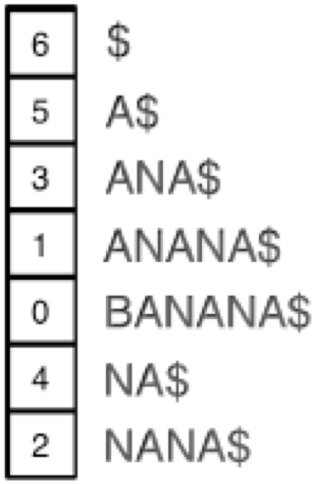
?

Indexing a genome

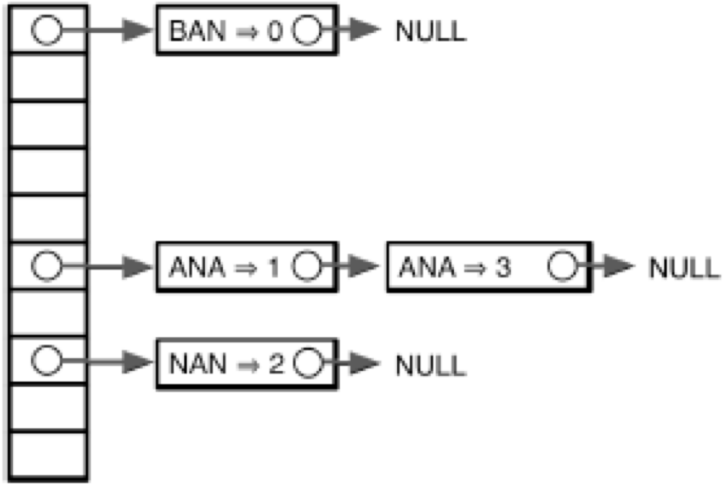
- To find exactly matching substrings, we need to build an index for the whole genome.
- Problem: The genome is BIG!
- Genome indices can be big. For human:



> 35 GBs



> 12 GBs

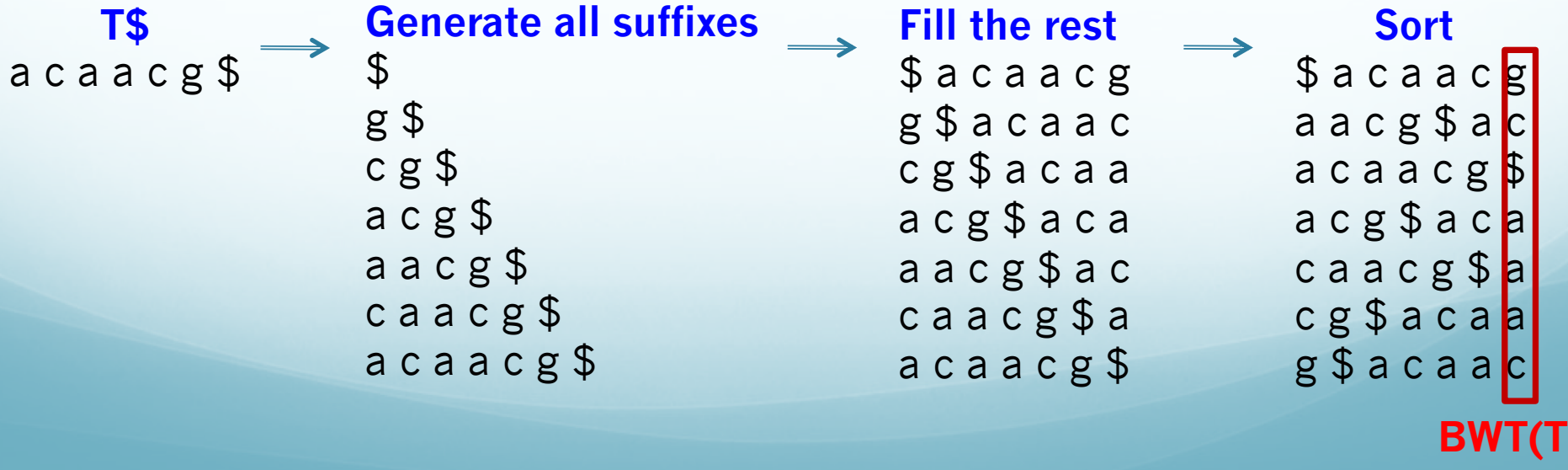


> 12 GBs

Burrows-Wheeler transform

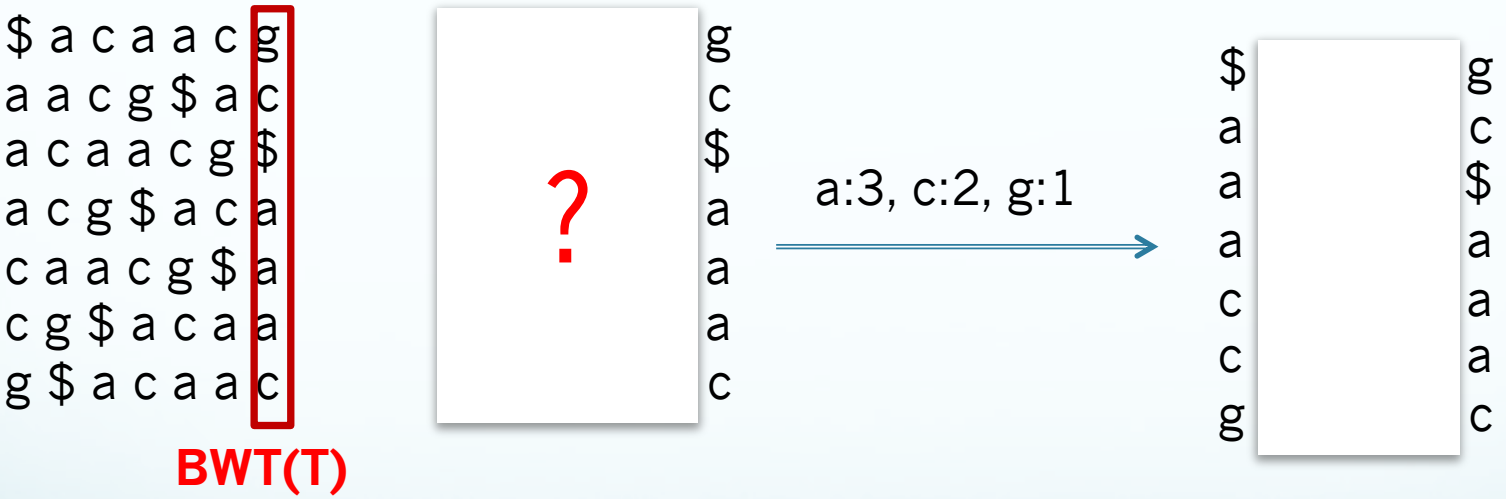
- http://en.wikipedia.org/wiki/Burrows-Wheeler_transform
- Reversible permutation used originally in compression
- T = a c a a c g

Burrows Wheeler Matrix



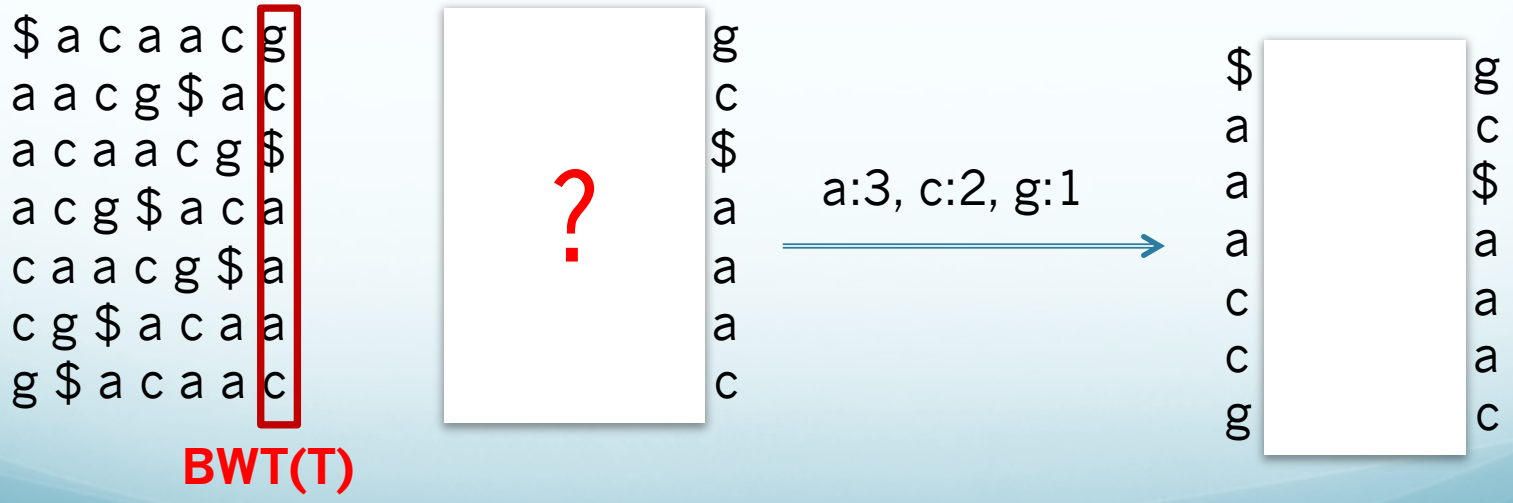
Burrows-Wheeler transform

- Can we reconstruct the first column with BWT(T)?



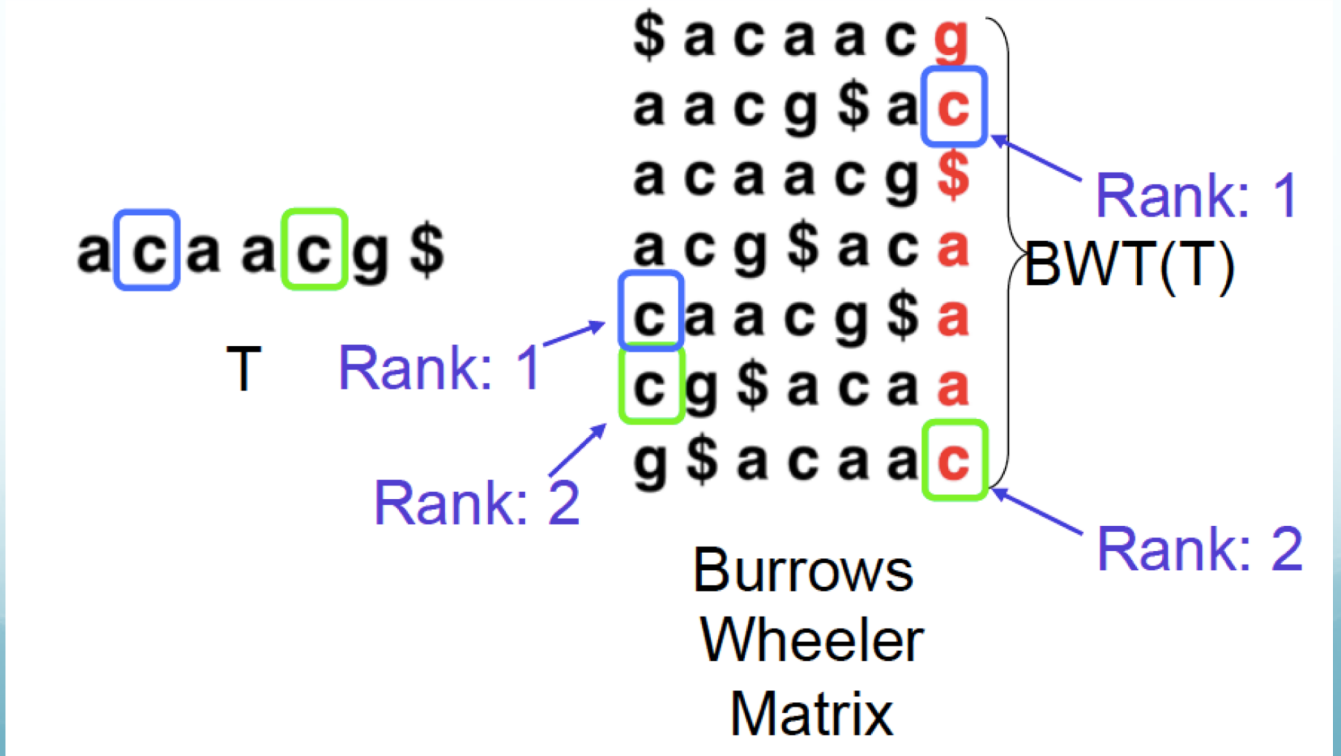
Burrows-Wheeler transform

- Can we reconstruct the first column with BWT(T)?
  - First column can be recovered by counting symbols in last column because it is sorted.



Burrows-Wheeler transform

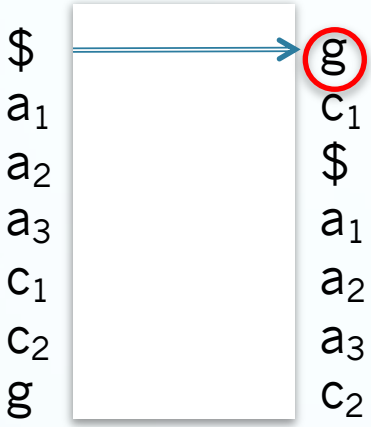
- Property that makes BWT(T) reversible is “LF Mapping”
  - ith occurrence of a character in Last column is same text occurrence as the ith occurrence in First column.



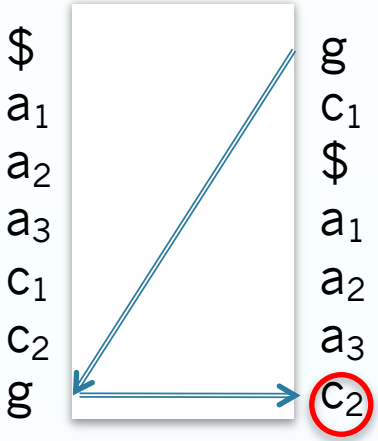
Burrows-Wheeler transform

- Can we reconstruct the original sequence, T = a c a a c g , using BTW(T)?
- We can recover the first column as described in the previous slide.
- T = a c a a c g ← reconstruct in the reverse order using suffix

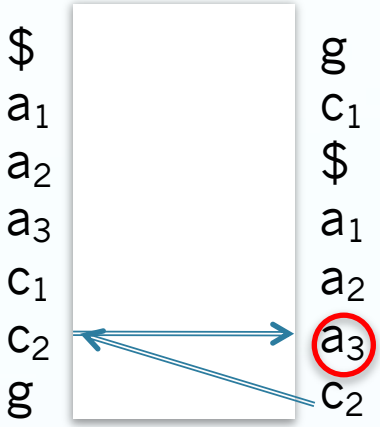
Burrows-Wheeler transform



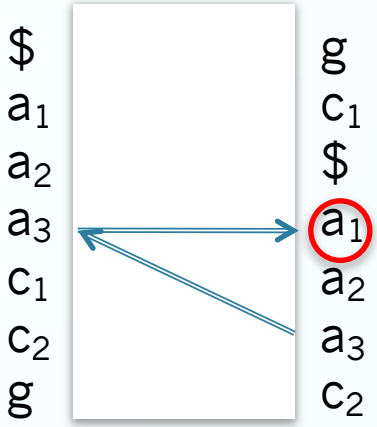
T = g



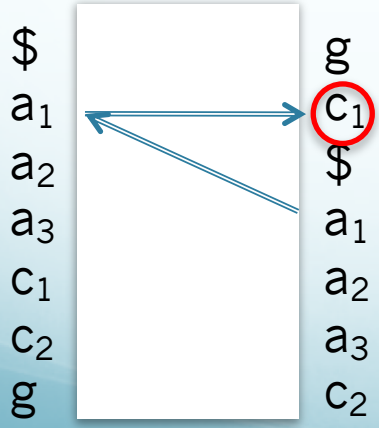
T = c_2 g



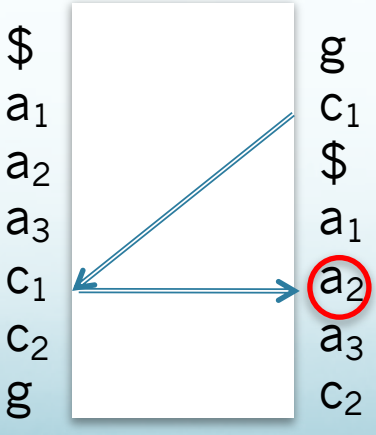
T = a_3 c_2 g



T = a_1 a_3 c_2 g



T = c_1 a_1 a_3 c_2 g



T = a_2 c_1 a_1 a_3 c_2 g

T = a c a a c g

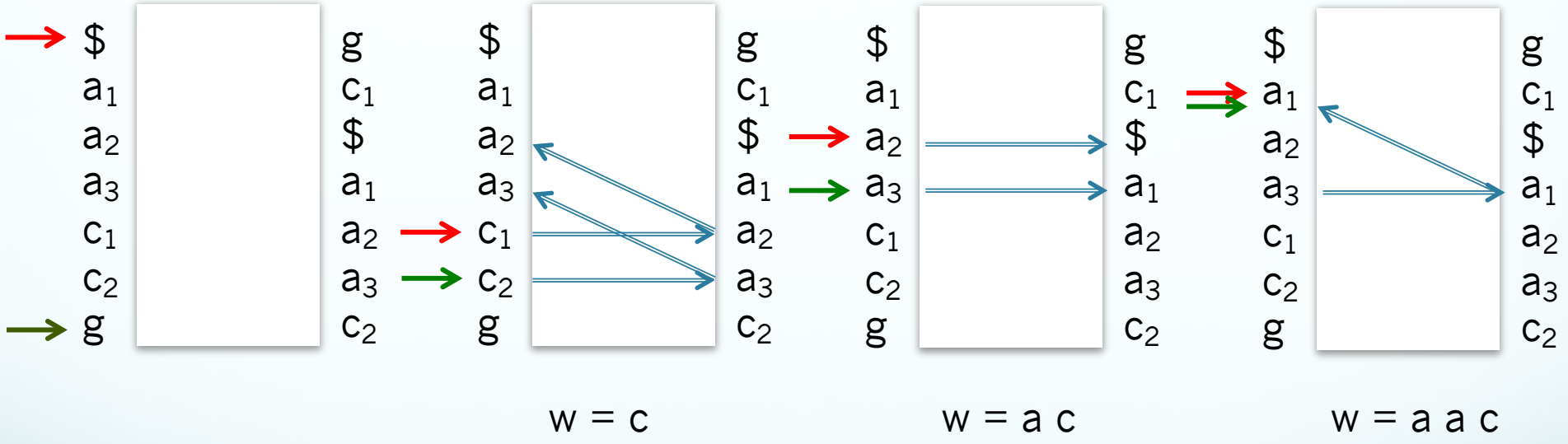
Exact Matching using BWT(T)

- 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 BTW(T) = g c $ a a a c, find pattern “a a c”

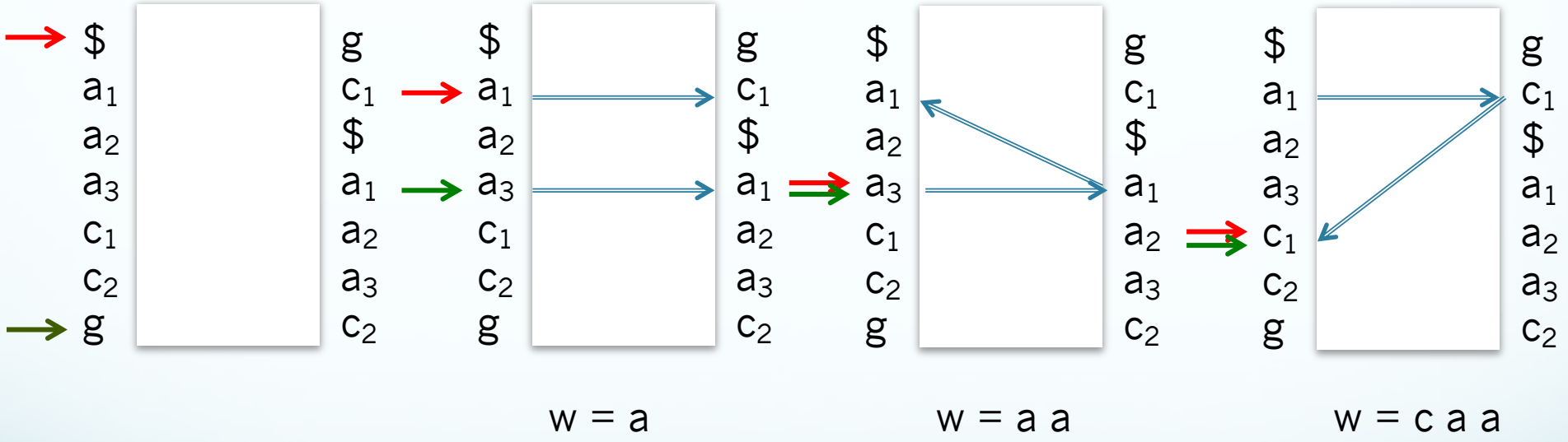
Exact Matching using BWT(T)

- Example T = a c a a c g, find pattern w = a a c



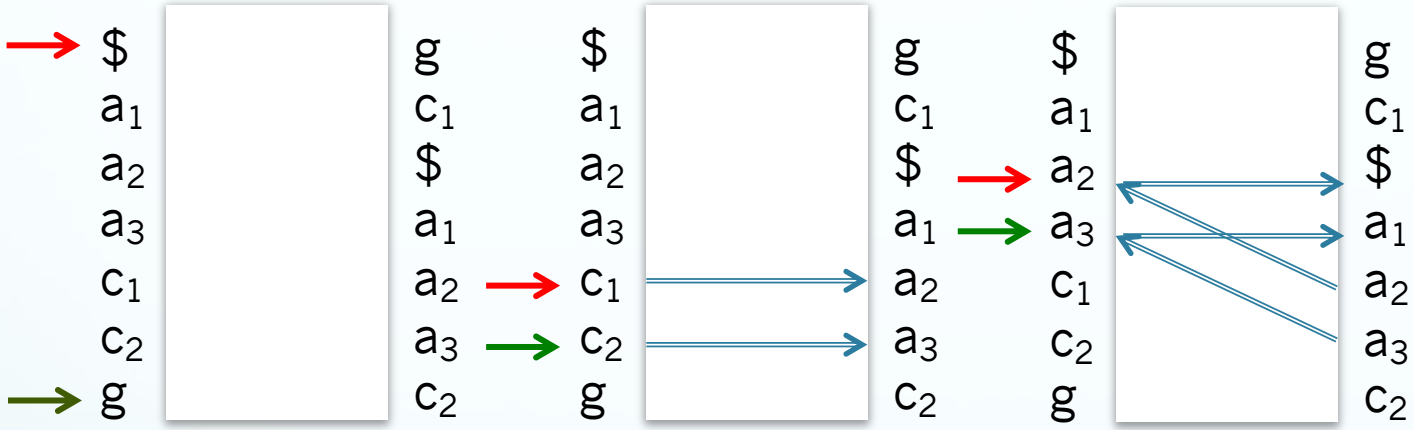
Exact Matching using BWT(T)

- Example T = a c a a c g, find pattern w = c a a



Exact Matching using BWT(T)

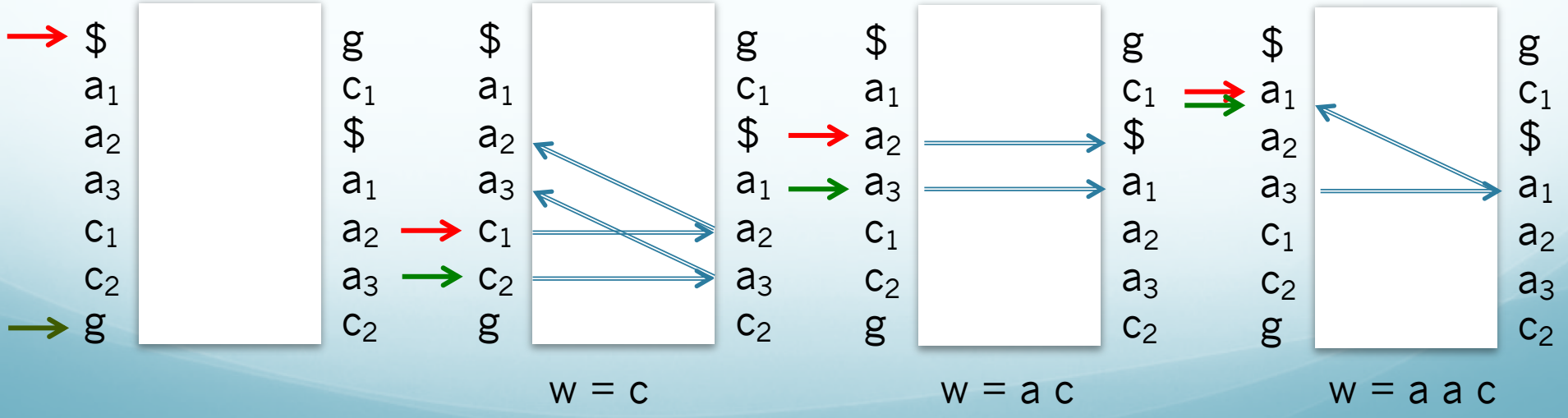
- Example T = a c a a c g, find pattern w = g a c



- pointers get lost.
- This way we can quickly check whether there is a match or not.

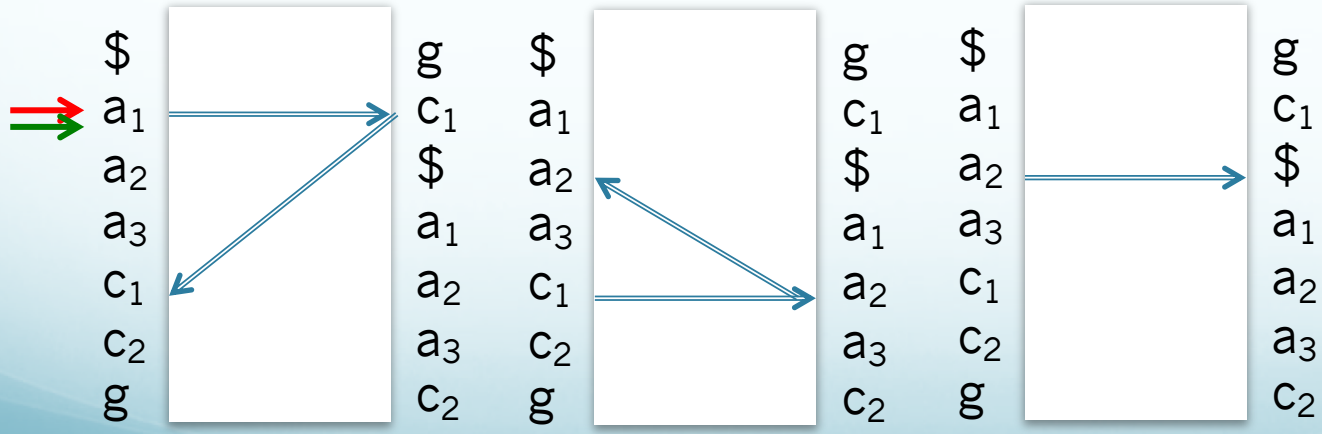
Where in sequence is the match?

- Use “walk-left” to build sequence to start
- Count number of sequences to get to the starting position.
- w = a a c



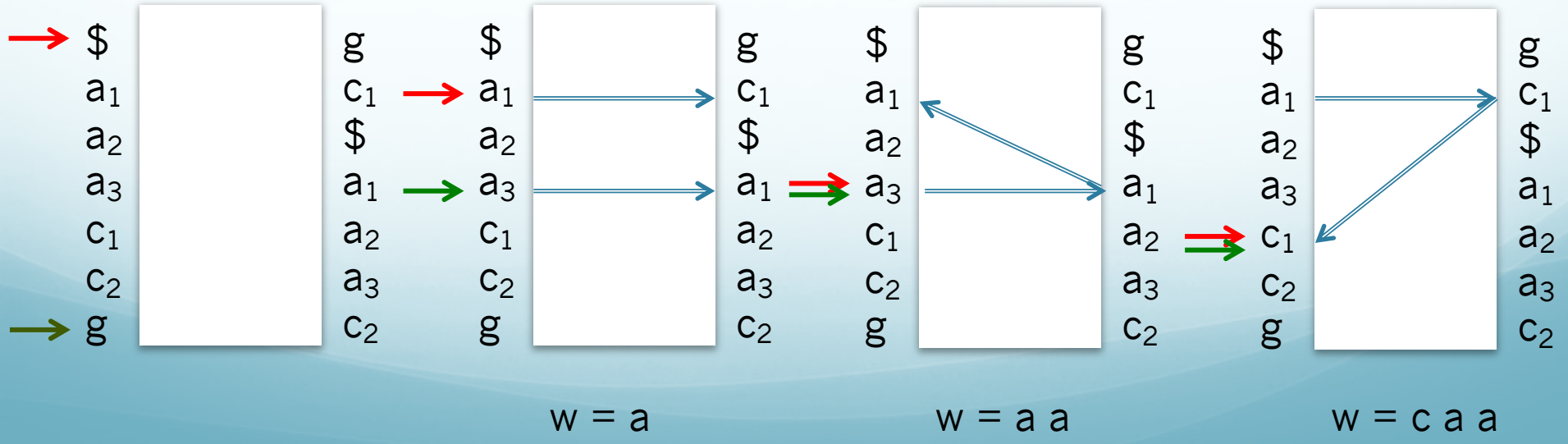
Where in sequence is the match?

- Use “walk-left” to build sequence to start
- Count number of sequences to get to the starting position.
- T = a c a a c g, w= a a c is in position 2



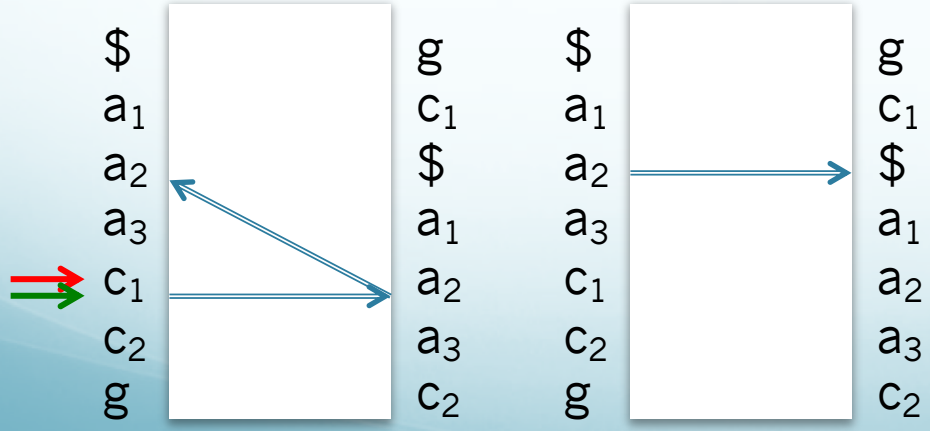
Where in sequence is the match?

- Use “walk-left” to build sequence to start
- Number of moves back to the starting position
- w = c a a



Where in sequence is the match?

- Use “walk-left” to build sequence to start
- Number of moves back to the starting position
- T = a c a a c g, w = c a a is in position 1

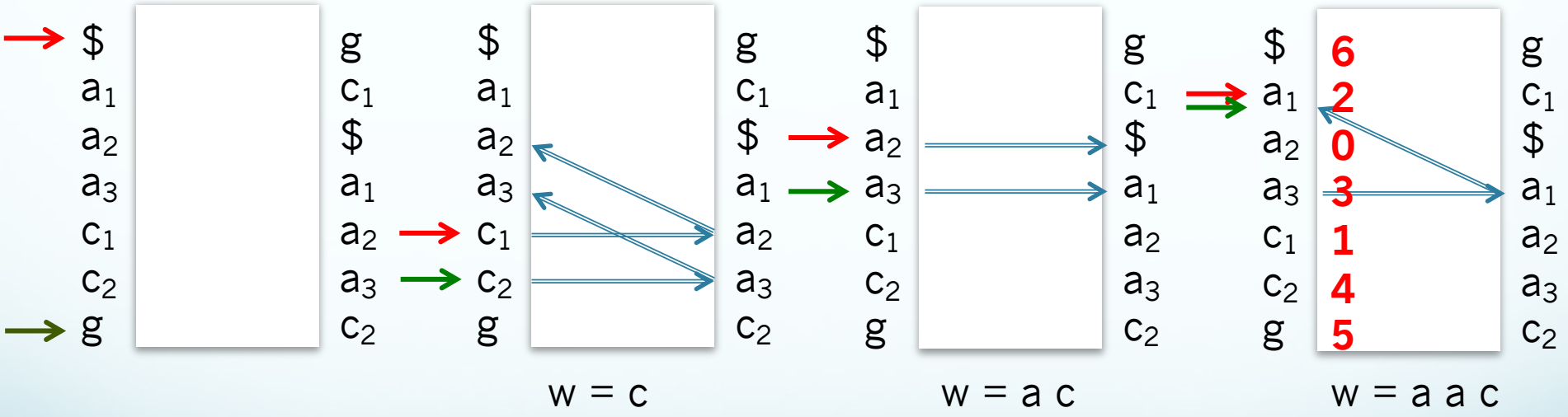


a c a a

- "walk-left" to start of sequence is slow
  - Requires on average N/2 steps to reach start.
  - Alternate strategy: keep index of positions.
  - T = a c a a c g

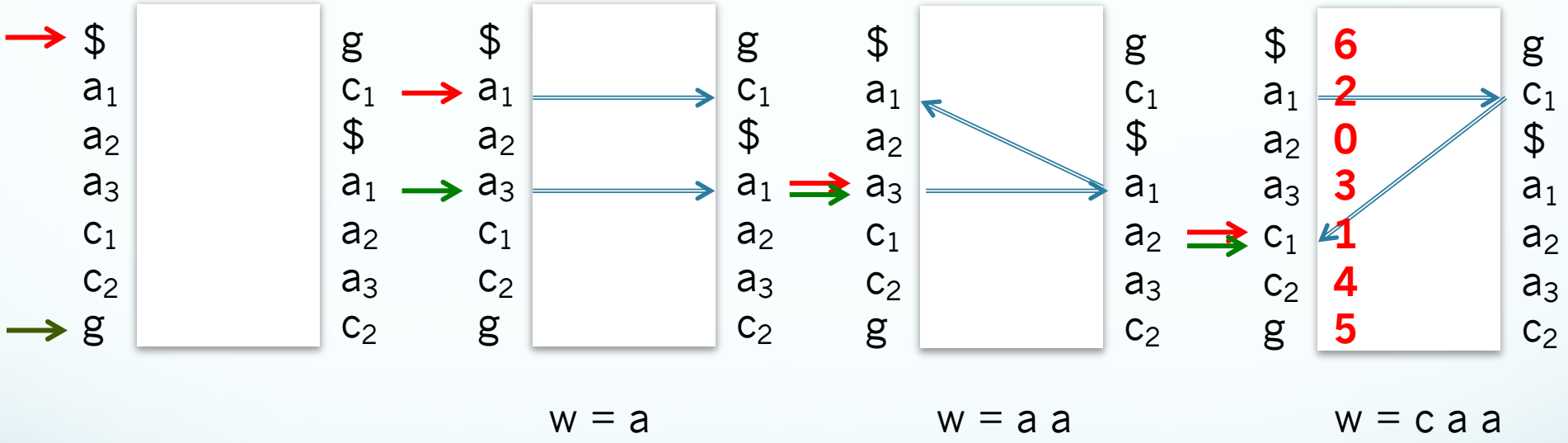
Where in sequence is the match?

- T = a c a a c g, w= a a c is in position 2



Exact Matching using BWT(T)

- T = a c a a c g, w = c a a in position 1

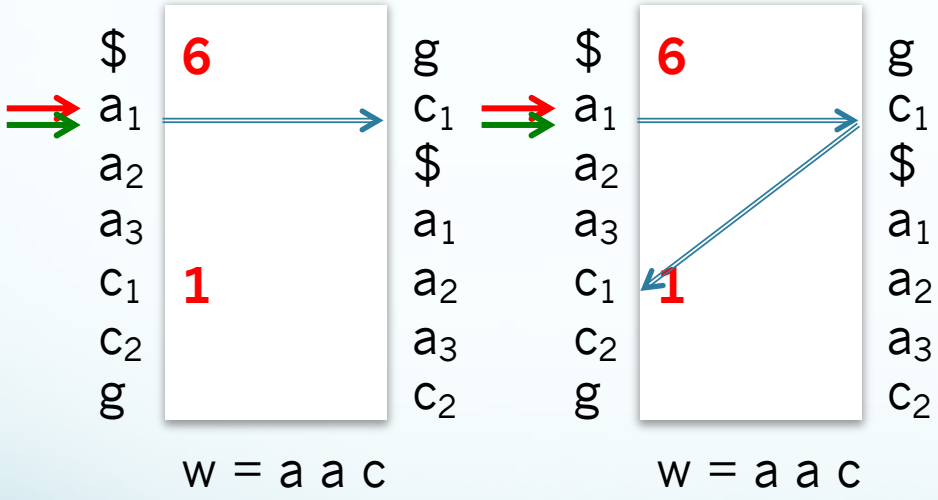


walk-left optimization

- we can save spae for index
- Key Idea: Store fraction of array (sampling)
- Only store some positions and “walk-left”
- Combines two previous strategies.
- How many values to store provides defines time/space tradeoff.

walk-left optimization

- T = a c a a c g, w= a a c is in position 2



Position =
number of moves + position in array
For “aac” = 1 + 1 = 2

BWT efficiency

- BWT is very fast

Table 1

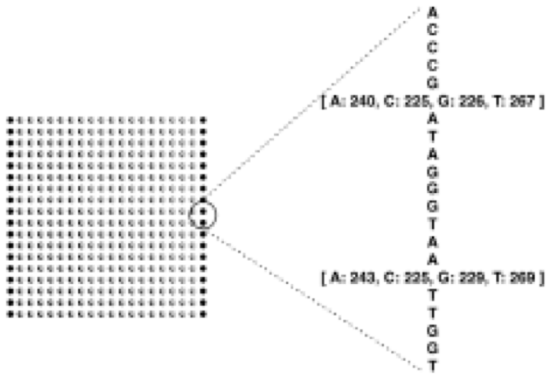
Bowtie alignment performance versus SOAP and Maq

	Platform	CPU time	Wall clock time	Reads mapped per hour (millions)	Peak virtual memory footprint (megabytes)	Bowtie speed-up	Reads aligned (%)
Bowtie -v 2	Server	15 m 7 s	15 m 41 s	33.8	1,149	-	67.4
SOAP		91 h 57 m 35 s	91 h 47 m 46 s	0.10	13,619	351×	67.3
Bowtie	PC	16 m 41 s	17 m 57 s	29.5	1,353	-	71.9
Maq		17 h 46 m 35 s	17 h 53 m 7 s	0.49	804	59.8×	74.7
Bowtie	Server	17 m 58 s	18 m 26 s	28.8	1,353	-	71.9
Maq		32 h 56 m 53 s	32 h 58 m 39 s	0.27	804	107×	74.7

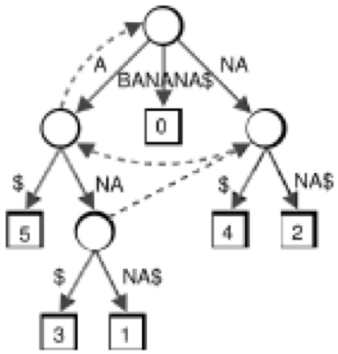
The performance and sensitivity of Bowtie v0.9.6, SOAP v1.10, and Maq v0.6.6 when aligning 8.84 M reads from the 1,000 Genome project (National Center for Biotechnology Information Short Read Archive: SRR001115) trimmed to 35 base pairs. The 'soap.contig' version of the SOAP binary was used. SOAP could not be run on the PC because SOAP's memory footprint exceeds the PC's physical memory. For the SOAP comparison, Bowtie was invoked with '-v 2' to mimic SOAP's default matching policy (which allows up to two mismatches in the alignment and disregards quality values). For the Maq comparison Bowtie is run with its default policy, which mimics Maq's default policy of allowing up to two mismatches during the first 28 bases and enforcing an overall limit of 70 on the sum of the quality values at all mismatched positions. To make Bowtie's memory footprint more comparable to Maq's, Bowtie is invoked with the '-z' option in all experiments to ensure only the forward or mirror index is resident in memory at one time. CPU, central processing unit.

BWT efficiency

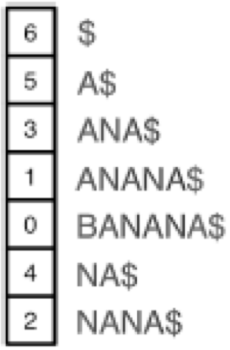
- BWT requires very small memory
- Total: ~1.65x the size of T
- Other methods



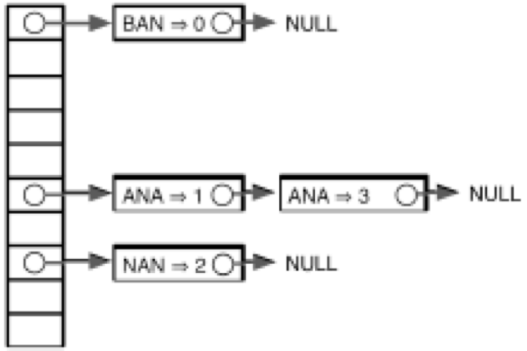
~1.65x



>45x



>15x



>15x

For human: 2.2.GB >35GB >12GB >12GB

참고자료

Hamiltonian path VS Eulerian path

- Notation
  - Input : A set S, representing all l-mers from an unknown string s
  - Output: String s such that Spectrum(s, l) = S
  - S = Spectrum(s, l)
  - $|S| = |s| - l + 1$
- Hamiltonian path
  - vertex: all l-mers
  - edge: direct edge(p,q) between p and q if last l-1 letters of p coincide with first l-2 letters of q
  - find a path that visits every "vertex" only once
  - time: construct graph + find path = $O(|S|^2) + O(2^{|s|}) = O(2^{|s|})$
  - NP-complete problem (between polynomial and exponential in worst case)
  - may not visit every edge

Hamiltonian path VS Eulerian path

- Notation
  - Input : A set S, representing all l-mers from an unknown string s
  - Output: String s such that Spectrum(s, l) = S
  - S = Spectrum(s, l)
  - |S| = |s|- l + 1
- Eulerian path
  - vertex: all (l-1)-mers
  - edge: direct edge(p,q) between p and q if spectrum(s,l) contains l-mers for which first l-1 letters coincide with p and last l-1 letters coincide with q
  - find a path that visits every “edge” only once
  - time: construct graph + find path = O(|S|) + O(|S|) = O(|S|), linear
  - may visit some vertices more than once

Summary of Eulerian path

- A connected graph is Eulerian if and only if each of its vertex is balanced (indegree=outdegree) except for start (1+indegree=outdegree) and end (indegree=a+outdegree) vertex.
- Find Eulerian cycle
  - Start from arbitrary vertex v, traverse univisited edges until meet a vertex which has no unvisited outgoing edges.
  - If the path is Eulerian, stop
  - Else it should contain a vertex w that has untraversed edges. Start from w, traverse unvisited edges until meet a vertex until meet a vertex which has no unvisited outgoing edge. compile two pathes (traverse the first path from v to w, traverse the second path from w to w itself, then traverse the remaining first path from w to v) : linear time algorithm.
  - Repeat the process above if there is any left edges.

references

- An Eulerian path approach to DNA fragment assembly, Pevzner et al, PNAS, 2011
- http://www.homolog.us/Tutorials/index.php?p=1.1&s=1
- http://www.cs.jhu.edu/~langmea/resources/lecture_notes/assembly_dbg.pdf