RNA-Sequencing

Nicolas Balcazar Corinna Blasse An Duc Dang
Hannes Hauswedell Sebastian Thieme

Advanced Algorithms for Bioinformatics (P4)
K. Reinert and S. Andreotti

SoSe 2010
FU Berlin

May 13, 2010

Outline

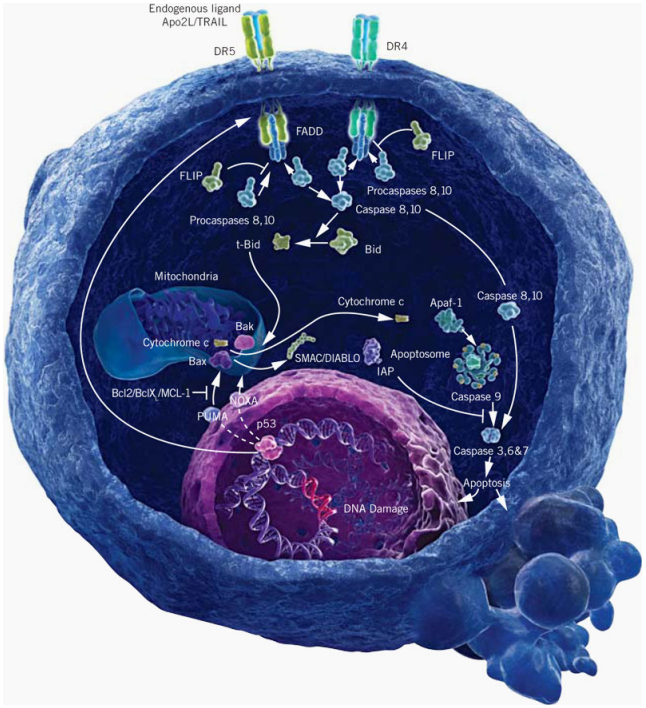
What is RNA-Seq?

- Also called "Whole Transcriptome Shotgun Sequencing"
- Technique to sequence cDNA in order to get information about a RNA sample content

Outline

Why do we use RNA-Seq?

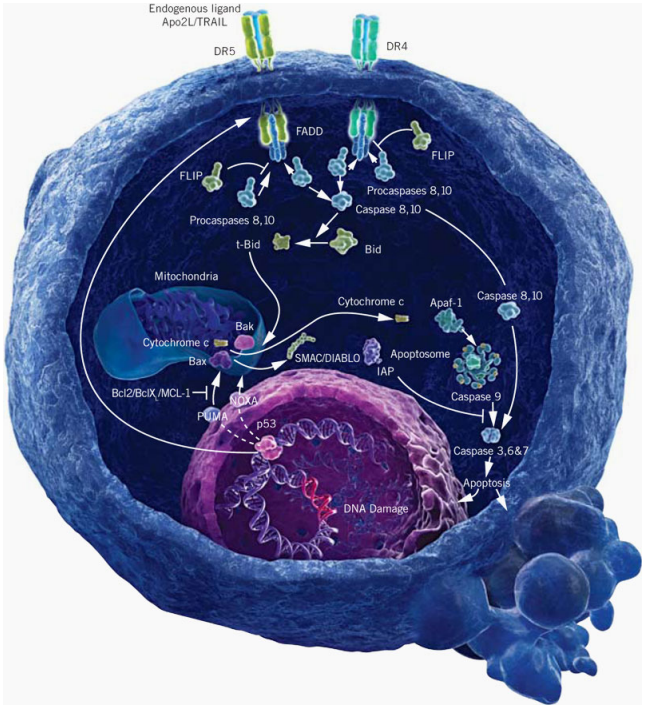
- Crucial to understand the processes inside a cell
- DNA cannot give us all information unlike transcriptomes
  - mRNAs specify cells' identity
  - mRNAs govern cells' activity due to environmental conditions



apoptosis pathway

Why do we use RNA-Seq?

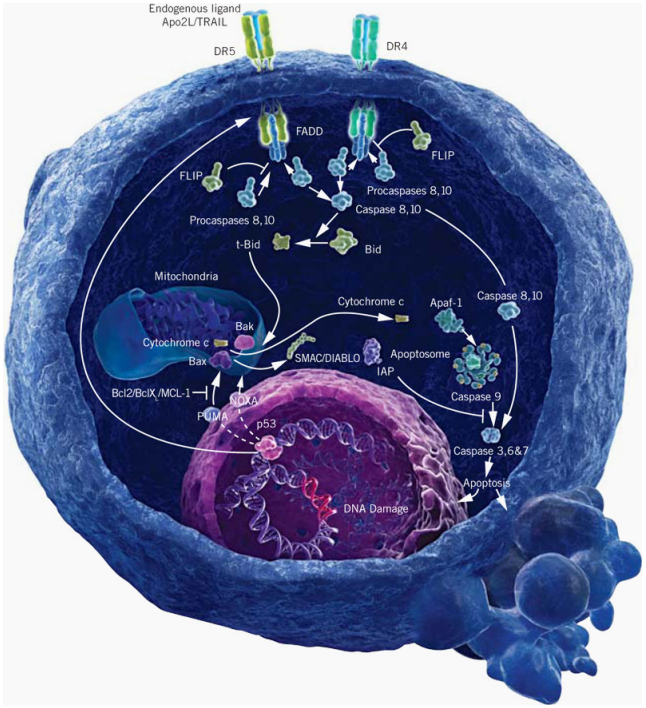
- Crucial to understand the processes inside a cell
- DNA cannot give us all information unlike transcriptomes
  - mRNAs specify cells' identity
  - mRNAs govern cells' activity due to environmental conditions



apoptosis pathway

Why do we use RNA-Seq?

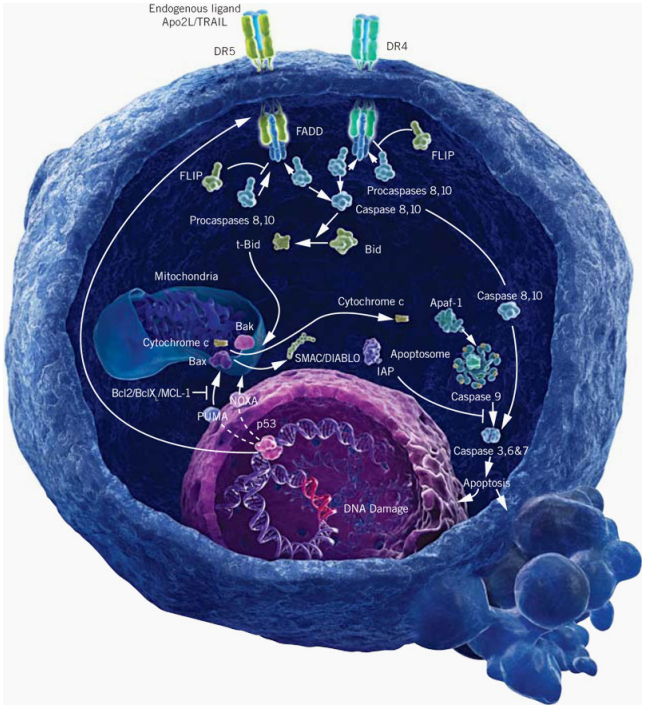
- Crucial to understand the processes inside a cell
- DNA cannot give us all information unlike transcriptomes
  - mRNAs specify cells' identity
  - mRNAs govern cells' activity due to environmental conditions



apoptosis pathway

Why do we use RNA-Seq?

- Crucial to understand the processes inside a cell
- DNA cannot give us all information unlike transcriptomes
  - mRNAs specify cells' identity
  - mRNAs govern cells' activity due to environmental conditions



apoptosis pathway

Advantages

- High-throughput sequencing
- No bacterial cloning of cDNA needed (no bacterial cloning constraints)
- High coverage
- Can discover new exons, genes
- Can handle RNA splicing
- Low cost

Advantages

- High-throughput sequencing
- No bacterial cloning of cDNA needed (no bacterial cloning constraints)
- High coverage
- Can discover new exons, genes
- Can handle RNA splicing
- Low cost

Advantages

- High-throughput sequencing
- No bacterial cloning of cDNA needed (no bacterial cloning constraints)
- High coverage
- Can discover new exons, genes
- Can handle RNA splicing
- Low cost

Advantages

- High-throughput sequencing
- No bacterial cloning of cDNA needed (no bacterial cloning constraints)
- High coverage
- Can discover new exons, genes
- Can handle RNA splicing
- Low cost

Advantages

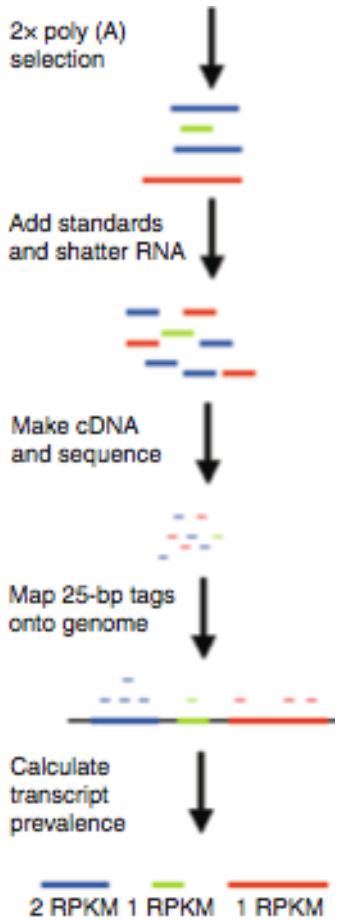
- High-throughput sequencing
- No bacterial cloning of cDNA needed (no bacterial cloning constraints)
- High coverage
- Can discover new exons, genes
- Can handle RNA splicing
- Low cost

Advantages

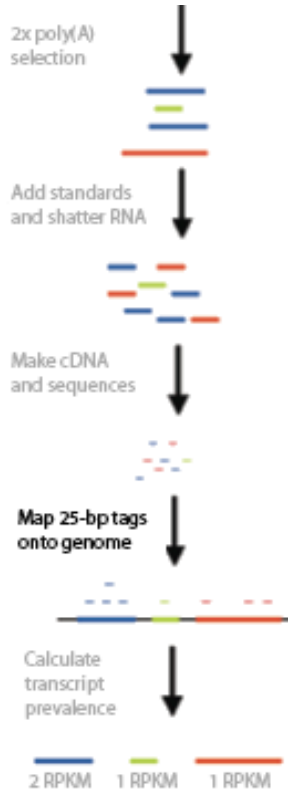
- High-throughput sequencing
- No bacterial cloning of cDNA needed (no bacterial cloning constraints)
- High coverage
- Can discover new exons, genes
- Can handle RNA splicing
- Low cost

Pipeline

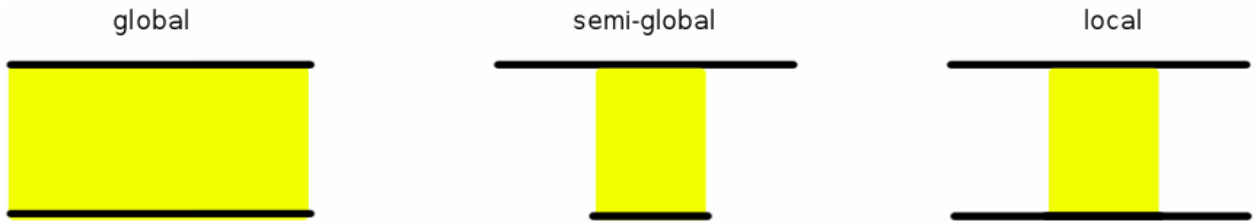
- Poly(A)-selection of RNA
- Fragmentation of RNA to an average length
- Conversion into cDNA
- Sequencing
- Mapping of the reads onto the genome
- Calculation of the transcript prevalence



Mapping



- *Read-mapping* is the semi-global alignment of many (very) short sequences (reads) to a long sequence (the genome)
- Alignment: approximate string matching



Important Questions

Questions regarding choice of algorithm and implementation:

Input
- How long are the reads? How many are there?
- How big is the genome?
- What technology was used?
- Known error-profile or quality values?

Output
- Gapped or ungapped alignments?
- What kind of scoring / distance measurement?

Important Questions

Questions regarding choice of algorithm and implementation:

Input
- How long are the reads? How many are there?
- How big is the genome?
- What technology was used?
- Known error-profile or quality values?

Output
- Gapped or ungapped alignments?
- What kind of scoring / distance measurement?

Tools

Tools designed for short sequences
(limited number of mismatches, short gaps)

- Bowtie
- RazerS
- SOAP
- MAQ
- ZOOM

Overview

Two main phases

1. Filtration / non Filtration (other preprocessing)
2. Verification

$\Rightarrow$ Different (combinations of) algorithms available

Overview

Two main phases

1. Filtration / non Filtration (other preprocessing)
2. Verification

$\Rightarrow$ Different (combinations of) algorithms available

Outline - Lectures

Lecture 1 Non filtering algorithm

Bowtie

- BWT

Lecture 2 Filtering algorithm

Filtering

- Pidgeonhole
- q-gram

RazerS

Lecture 3 Repeat Resolution

Outline - Lectures

Lecture 1 Non filtering algorithm

Bowtie

- BWT

Lecture 2 Filtering algorithm

Filtering

- Pidgeonhole
- q-gram

RazerS

Lecture 3 Repeat Resolution

Outline - Lectures

Lecture 1 Non filtering algorithm

Bowtie

- BWT

Lecture 2 Filtering algorithm

Filtering

- Pidgeonhole
- q-gram

RazerS

Lecture 3 Repeat Resolution

The paper

Software

Open Access

Ultrafast and memory-efficient alignment of short DNA sequences to the human genome

Ben Langmead, Cole Trapnell, Mihai Pop and Steven L Salzberg

Address: Center for Bioinformatics and Computational Biology, Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA.

Correspondence: Ben Langmead. Email: langmead@cs.umd.edu

Published: 4 March 2009

Genome ***Biology*** 2009, **10:**R25 (doi:10.1186/gb-2009-10-3-r25)

The electronic version of this article is the complete one and can be found online at http://genomebiology.com/2009/10/3/R25

Received: 21 October 2008
Revised: 19 December 2008
Accepted: 4 March 2009

Abstract

Bowtie is an ultrafast, memory-efficient alignment program for aligning short DNA sequence reads to large genomes. For the human genome, Burrows-Wheeler indexing allows Bowtie to align more than 25 million reads per CPU hour with a memory footprint of approximately 1.3 gigabytes

Bowtie

- Fast, memory-efficient
- Burrows-Wheeler indexing → align >25 million reads per CPU hour (memory footprint aprox. 1.3 GB)
- Extends previous Burrows-Wheeler techniques with a novel quality-aware backtracking algorithm that permits mismatches
- Mismatches → Backtracking

Bowtie

- Fast, memory-efficient
- Burrows-Wheeler indexing $\rightarrow$ align >25 million reads per CPU hour (memory footprint aprox. 1.3 GB)
- Extends previous Burrows-Wheeler techniques with a novel quality-aware backtracking algorithm that permits mismatches
- Mismatches $\rightarrow$ Backtracking

Bowtie

- Fast, memory-efficient
- Burrows-Wheeler indexing $\rightarrow$ align >25 million reads per CPU hour (memory footprint aprox. 1.3 GB)
- Extends previous Burrows-Wheeler techniques with a novel quality-aware backtracking algorithm that permits mismatches
- Mismatches $\rightarrow$ Backtracking

Bowtie

- Fast, memory-efficient
- Burrows-Wheeler indexing $\rightarrow$ align >25 million reads per CPU hour (memory footprint aprox. 1.3 GB)
- Extends previous Burrows-Wheeler techniques with a novel quality-aware backtracking algorithm that permits mismatches
- Mismatches $\rightarrow$ Backtracking

Bowtie

Goal? → map reads back to reference genome

- Exact match
- Inexact match

Bowtie

Goal? $\rightarrow$ map reads back to reference genome

- Exact match
- Inexact match

Bowtie

Goal? $\rightarrow$ map reads back to reference genome

- Exact match
- Inexact match

Exact Match

→ Exact string matching → EXACTMATCH

Inexact Match

- Mismatches or indels
- Backtracking

Efficiency

- compress genome $\rightarrow$ space efficient
- indexation of genome $\rightarrow$ fast search
- ideas? $\rightarrow$ suffix array
- or even better $\rightarrow$ Burrows-Wheeler indexing (BWT)

Efficiency

- compress genome $\rightarrow$ space efficient
- indexation of genome $\rightarrow$ fast search
- ideas? $\rightarrow$ suffix array
- or even better $\rightarrow$ Burrows-Wheeler indexing (BWT)

Efficiency

- compress genome $\rightarrow$ space efficient
- indexation of genome $\rightarrow$ fast search
- ideas? $\rightarrow$ suffix array
- or even better $\rightarrow$ Burrows-Wheeler indexing (BWT)

Efficiency

- compress genome $\rightarrow$ space efficient
- indexation of genome $\rightarrow$ fast search
- ideas? $\rightarrow$ suffix array
- or even better $\rightarrow$ Burrows-Wheeler indexing (BWT)

Efficiency

- compress genome $\rightarrow$ space efficient
- indexation of genome $\rightarrow$ fast search
- ideas? $\rightarrow$ suffix array
- or even better $\rightarrow$ Burrows-Wheeler indexing (BWT)

Suffix Arrays

- array of *integers*
- gives the starting positions of suffixes of a string
- in lexicographical order
- very space economical data structure

Suffix Arrays

Example

Consider the string "'abracadabra$"'

1	2	3	4	5	6	7	8	9	10	11	12	index
a	b	r	a	c	a	d	a	b	r	a	$	string

It has twelve suffixes:
abracadabra$,
bracadabra$,
racadabra$,
...,
a$,
$

Suffix Arrays

Example

index	sorted suffix
12	$
11	a$
8	abra$
1	abracadabra$
4	acadabra$
6	adabra$
9	bra$
2	bracadabra$
5	cadabra$
7	dabra$
10	ra$
3	racadabra$

$\Rightarrow$ For the string "abracadabra$", the suffix array is [12, 11, 8, 1, 4, 6, 9, 2, 5, 7, 10, 3].

Burrows-Wheeler transformation

- published by Michael Burrows and David Wheeler in 1994
- reversible transformation algorithm to a input text
- does no data compression itself, but might lead to effective compression

Outline

Principle of BWT

- Input: **string** S of length N with characters of ordered alphabet Σ
- i.e.: $S = mississippi$, $N = 11$, $\Sigma = \{i, m, p, s\}$

- Output: **string** L of length $N + 1$ with characters of ordered alphabet $\Sigma \cup$ special character and index I
- i.e.: $L = ipssm\#pissii$, $I = 5$

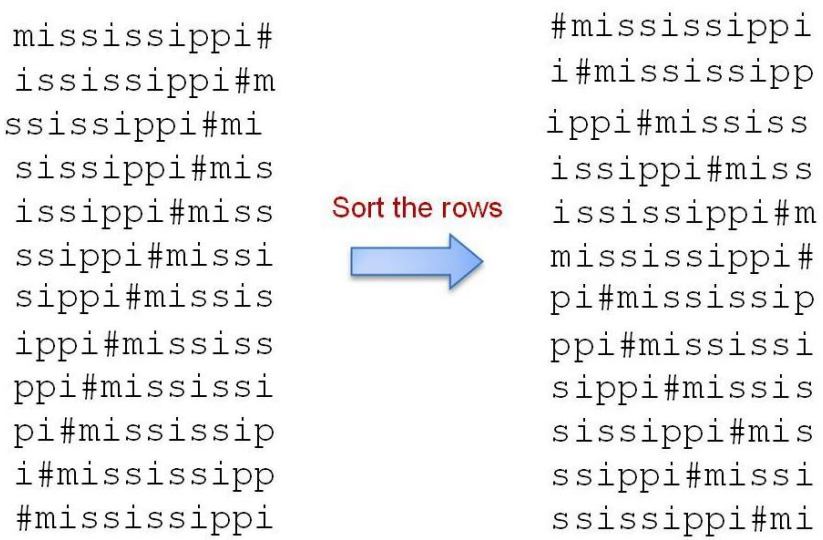
Principle of BWT

1. sort rotations
   - append an special character $\notin \Sigma$ like $\#$ to S, $\#$ is lexicographically minimal
   - build a conceptual $N + 1 \times N + 1$ Matrix M whose rows are cyclic shifts of S

```
mississippi#
ississippi#m
ssissippi#mi
sissippi#mis
issippi#miss
ssippi#missi
sippi#missis
ippi#mississ
ppi#mississi
pi#mississip
i#mississipp
#mississippi
```

Principle of BWT

1. sort rotations
   - append an special character $\notin \Sigma$ like $'\#'$ to S, $\#$ is lexicographically minimal
   - build a conceptual $N + 1 \times N + 1$ Matrix M whose rows are cyclic shifts of S
   - sort rows lexicographically



Principle of BWT

2. find last characters of rotations

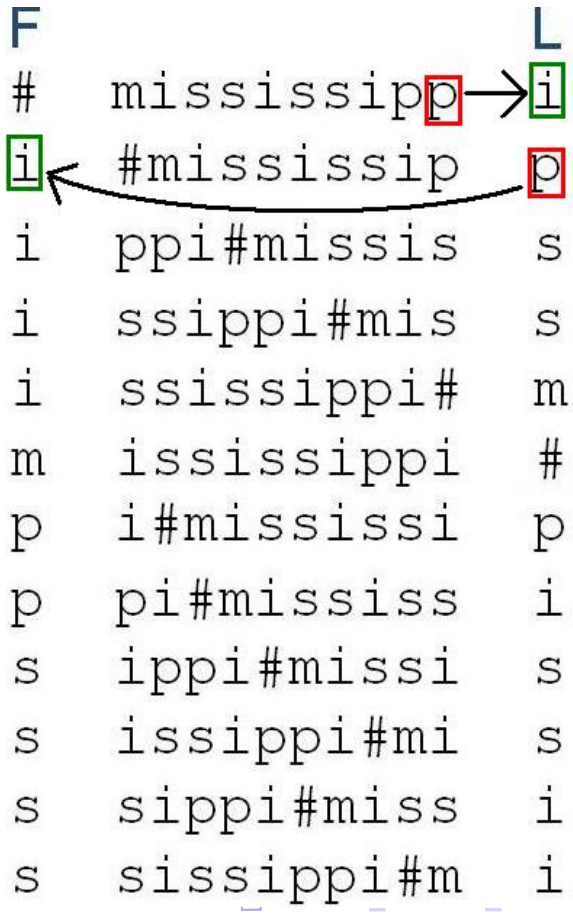
- string L or F is the last respectively first column of matrix M
  i.e.:
  $L = ipssm\#pissii$
- characters in $L[i]$ are predecessors of characters in $F[i]$

F		L
#	mississipp	i
i	#mississip	p
i	ppi#missis	s
i	ssippi#mis	s
i	ssissippi#	m
m	ississippi	#
p	i#mississi	p
p	pi#mississ	i
s	ippi#missi	s
s	issippi#mi	s
s	sippi#miss	i
s	sissippi#m	i

Principle of BWT

2. find last characters of rotations

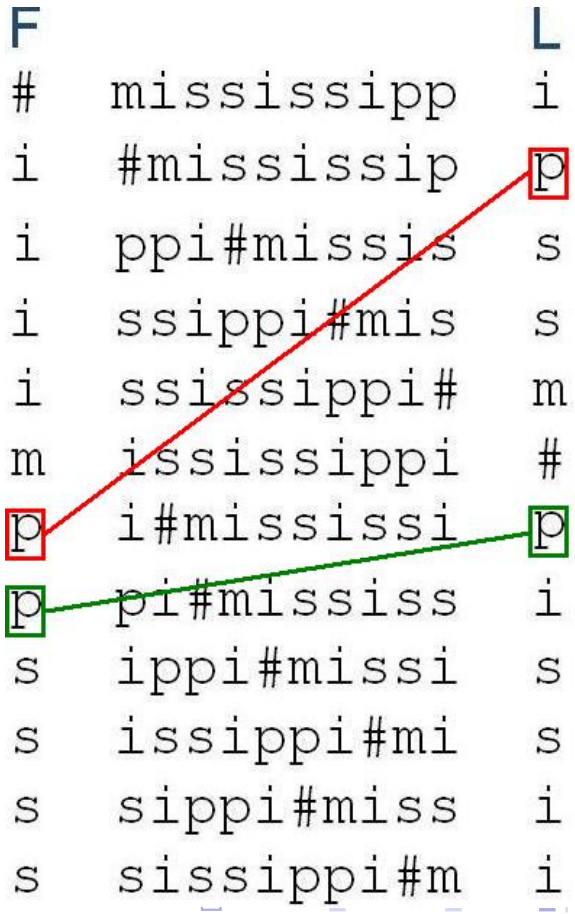
- string L or F is the last respectively first column of matrix M
  i.e.:
  $L = ipssm\#pissii$
- characters in $L[i]$ are predecessors of characters in $F[i]$



Principle of BWT

2. find last characters of rotations

- rank preserving property: $pos_L(c_1) < pos_L(c_2) \Leftrightarrow pos_F(c_1) < pos_F(c_2)$, whereas $pos_X(c)$: position of character c in string X
- index I is row number of input $S\cup$ special character (i.e. $I = 5$)



Principle of BWT

2. find last characters of rotations

- rank preserving property: $pos_L(c_1) < pos_L(c_2) \Leftrightarrow pos_F(c_1) < pos_F(c_2)$, whereas $pos_X(c)$: position of character c in string X
- index *I* is row number of input $S\cup$ special character (i.e. $I = 5$)

F		L
#	mississipp	i
i	#mississip	p
i	ppi#missis	s
i	ssippi#mis	s
i	ssissippi#	m
m	ississippi	#
p	i#mississi	p
p	pi#mississ	i
s	ippi#missi	s
s	issippi#mi	s
s	sippi#miss	i
s	sissippi#m	i

Principle of BWT

Fig. 7: Part of an BWT ouput example from the original BWT paper.

final char (L)	sorted rotations
a	n to decompress. It achieves compression
o	n to perform only comparisons to a depth
o	n transformation} This section describes
o	n transformation} We use the example and
o	n treats the right-hand side as the most
a	n tree for each 16 kbyte input block, enc
a	n tree in the output stream, then encodes
i	n turn, set $L[i]$ to be the
i	n turn, set $R[i]$ to the
o	n unusual data. Like the algorithm of Man
a	n use a single set of probabilities table
e	n using the positions of the suffixes in
i	n value at a given point in the vector $R
e	n we present modifications that improve t
e	n when the block size is quite large. Ho
i	n which codes that have not been seen in
i	n with ch appear in the {\em same order
i	n with ch. In our exam
o	n with Huffman or arithmetic coding. Bri
o	n with figures given by Bell~\cite{bell}.

Advantages of BWT

- due to many repetitions in a big input S string L has regions of same characters
- $\rightarrow$ good for compression algorithms (may be talked about next lecture)
- BWT stores string of characters instead of integer indexes like suffix arrays! $\rightarrow$ less space for big text input like DNA

Outline

Retransformation

- Input: string L of length $N + 1$ with characters of ordered alphabet $\Sigma \cup$ special character and index I
- i.e.: $L = ipssm\#pissii$, $N = 11$, $\Sigma = \{i, m, p, s\}$, special character $= \#$ and $I = 5$

- Output: string S of length N with characters of ordered alphabet Σ
- i.e.: $S =mississippi$

Retransformation

1. find F using L
   - every column in matrix M is permutation of string $S\#$
   - first column F and last column L also permutations of $S\#$ and of one another
   - M has sorted rows and F is first column $\Rightarrow$ F is sorted
   - get F by sorting L

F		L
#	mississipp	i
i	#mississip	p
i	ppi#missis	s
i	ssippi#mis	s
i	ssissippi#	m
m	ississippi	#
p	i#mississi	p
p	pi#mississ	i
s	ippi#missi	s
s	issippi#mi	s
s	sippi#miss	i
s	sissippi#m	i

Retransformation

1. find F using L
   - every column in matrix M is permutation of string $S\#$
   - first column F and last column L also permutations of $S\#$ and of one another
   - M has sorted rows and F is first column $\Rightarrow$ F is sorted
   - get F by sorting L

F	L
#	i
i	p
i	s
i	s
i	m
m	#
p	p
p	i
s	s
s	s
s	i
s	i

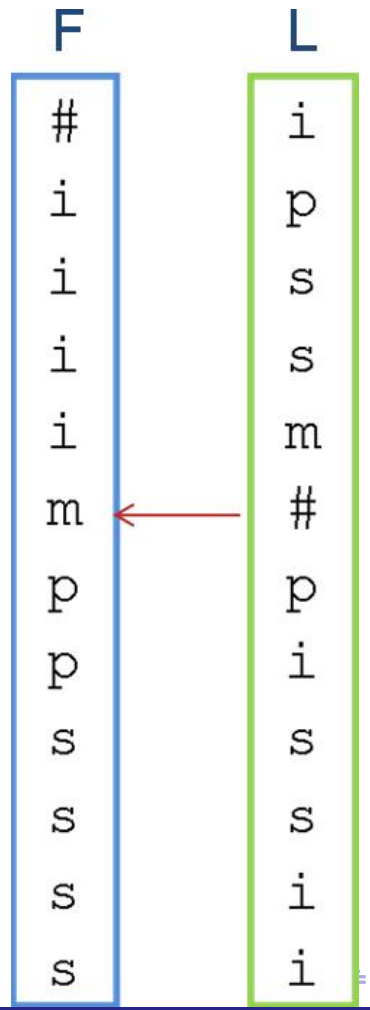
Retransformation

2. F to L mapping
   - build mapping vector T where $T[i]$ is position of the character in L corresponding to character $F[i]$ (bijective mapping)
   - i.e.: $T = \{5, 0, 7, 10, 11, 4, 1, 6, 2, 3, 8, 9\}$ (rank preserving!)

F	L
#	i
i	p
i	s
i	s
i	m
m	#
p	p
p	i
s	s
s	s
s	i
s	i

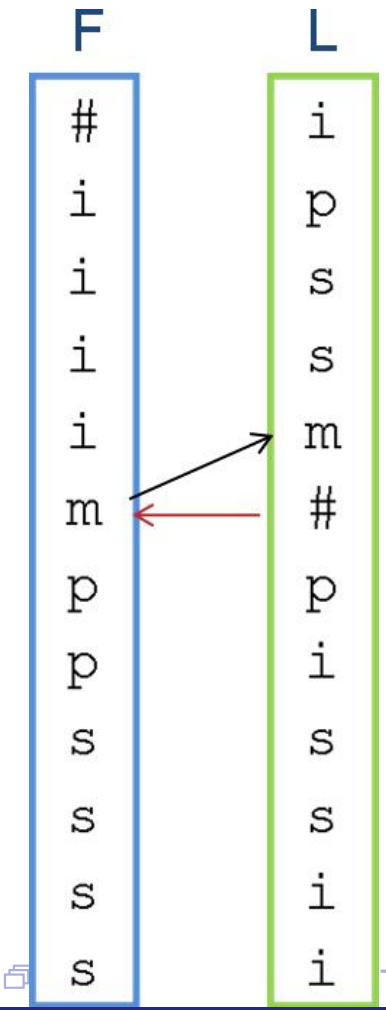
Retransformation

2. *F* to *L* mapping
   - recover *S* by starting at position of # in *L*, which is the last character in $S \cup \#$
   - successor of # is the character of *F* in the same row
   - i.e.: *m* is first character of *S*



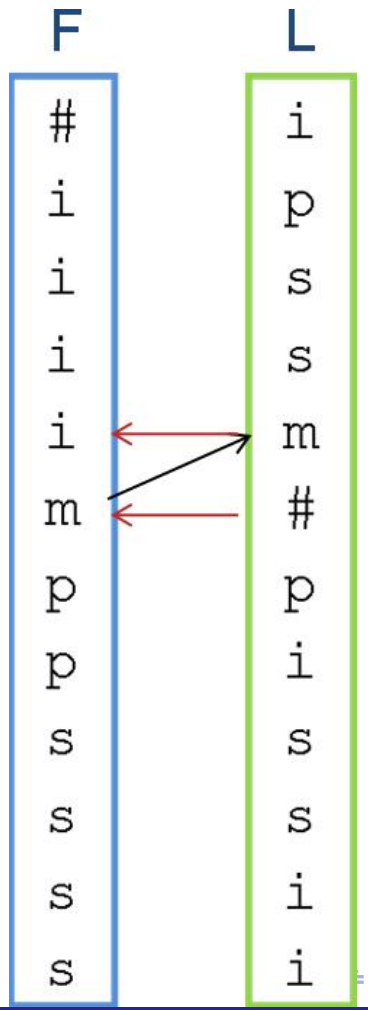
Retransformation

2. *F* to *L* mapping
   - use mapping vector *T* to get back from *m* in *F* to *m* in *L*
   - jump from last character of this row to first character to get next successor character of *S*
   - i.e.: we get *mi*



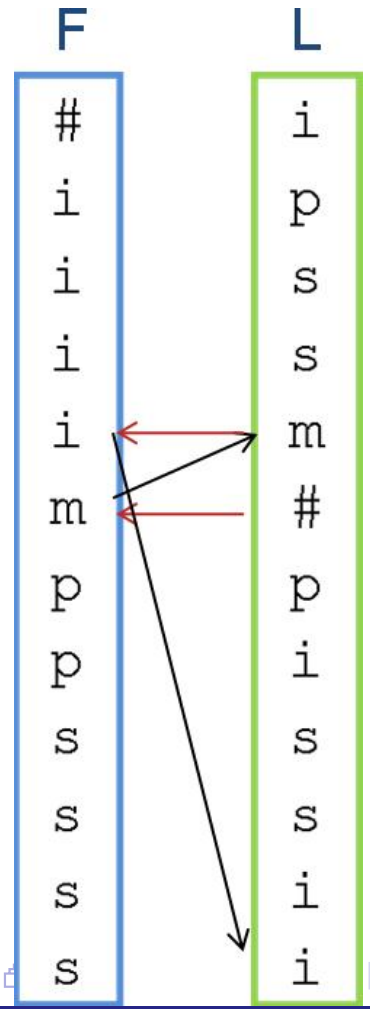
Retransformation

2. *F* to *L* mapping
   - use mapping vector *T* to get back from *m* in *F* to *m* in *L*
   - jump from last character of this row to first character to get next successor character of *S*
   - i.e.: we get *mi*



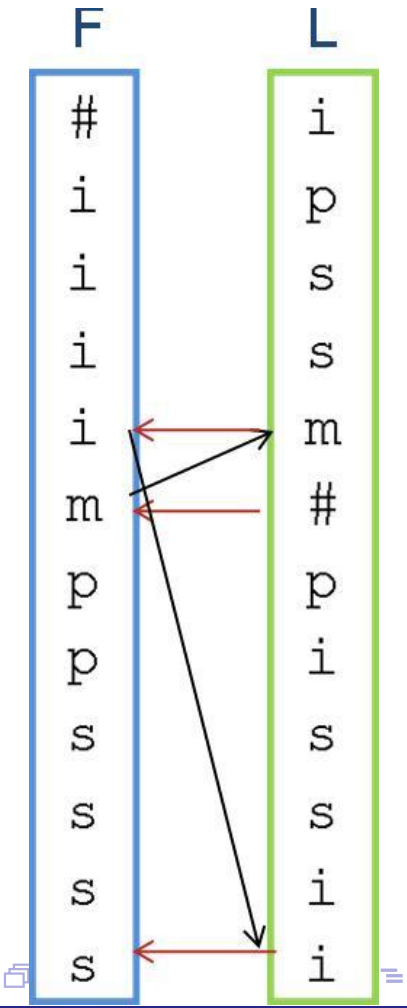
Retransformation

2. *F* to *L* mapping
   - use mapping vector *T* to get back from *i* in *F* to *i* in *L*
   - jump to front
   - i.e.: we get *mis*
   - go on until # in *F* reached



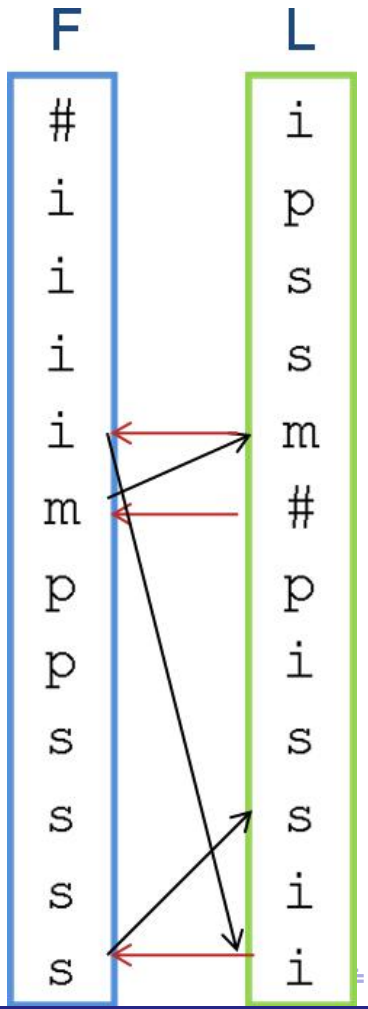
Retransformation

2. *F* to *L* mapping
   - use mapping vector *T* to get back from *i* in *F* to *i* in *L*
   - jump to front
   - i.e.: we get *mis*
   - go on until # in *F* reached



Retransformation

2. *F* to *L* mapping
   - use mapping vector *T* to get back from *i* in *F* to *i* in *L*
   - jump to front
   - i.e.: we get *mis*
   - go on until # in *F* reached



Where are we?

We have

- Reads
- Index of the reference genome (BWT)

We want

- Exact matches

si

psi

F		L
#	mississipp	i
i	#mississip	p
i	ppi#missis	s
i	ssippi#mis	s
i	ssissippi#	m
m	ississippi	#
p	i#mississi	p
p	pi#mississ	i
s	ippi#missi	s
s	issippi#mi	s
s	sippi#miss	i
s	sissippi#m	i

FM-Index

Full-Text Index In Minute Space

- Motivation: Finding reads by looking at only a small portion of the compressed text
- Use the relation between the Burrows-Wheeler compression algorithm and the suffix array data structure
- FM-Index = Compressed suffix array that connects both the compressed text and the full-text indexing information

- Supports two basic operations:
  - **Count** - return number of occurrences of P in T
  - **Locate** - find all positions of P in T

Outline

Backward-search algorithm

= Algorithm that counts the number of occurrences of P in T

- Input: L, C, Occ()
- Output: Number of occurrences

Example:

- si $\rightarrow$ 2
- pssi $\rightarrow$ 0

F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
s	ippi#missi	s	9
s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

Backward-search algorithm

- C[1,...,|Σ|]: C[c] contains the total number of chars in T which are alphabetically smaller than c (including char repetitions)

Example

- C[] for *mississippi#*

i	m	p	s

Backward-search algorithm

- C[1,...,|Σ|]: C[c] contains the total number of chars in T which are alphabetically smaller than c (including char repetitions)

Example

- C[] for *mississippi#*

i	m	p	s
I			

Backward-search algorithm

- C[1,...,|Σ|]: C[c] contains the total number of chars in T which are alphabetically smaller than c (including char repetitions)

Example

- C[] for *mississippi#*

i	m	p	s
1	5	6	8

Backward-search algorithm

- Occ(c,q): Number of occurrences of char c in prefix L[1,q]

Example

Occ(s,5) =

F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
s	ippi#missi	s	9
s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

Backward-search algorithm

- Occ(c,q): Number of occurrences of char c in prefix L[1,q]

Example

Occ(s,5) = 2

F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
s	ippi#missi	s	9
s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

Backward-search algorithm

- Occ(c,q): Number of occurrences of char c in prefix L[1,q]

Example

Occ(s,5) = 2
Occ(s,12) =

F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
s	ippi#missi	s	9
s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

Backward-search algorithm

- Occ(c,q): Number of occurrences of char c in prefix L[1,q]

Example

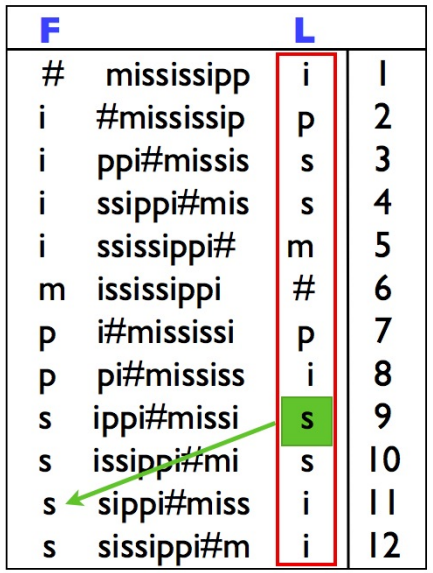
Occ(s,5) = 2
Occ(s,12) = 4

F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
s	ippi#missi	s	9
s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

L-to-F Mapping

- Formula to determine the position of char L[i] in F:
  pos(i) = C[L[i]] + Occ(L[i], i)
- Why?
  - C[L[i]] = Number of characters before the first L[i] in F
  - Occ(L[i], i) = Occurrence of character L[i] in string L[1,i]
- Example: i=9

$$pos(9) = C[s] + Occ(s, 9)$$
$$= 8 + 3 = 11$$



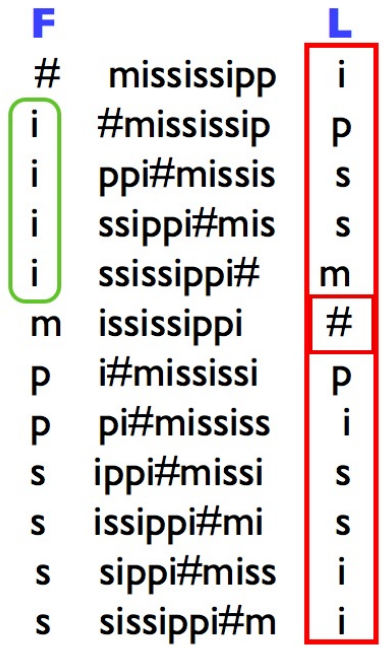
i	m	p	s
1	5	6	8

Occ(s, 9)= 3

Backward-search algorithm

How do we find a pattern of length = 1?

- Remember: BWT matrix rows = sorted suffixes of T
  - All suffixes prefixed by P occupies in a set of rows
  - Possible to calculate the first and last position
  - Occurrences = Last - First +1
- How do we search?



Backward-search algorithm

How do we find a pattern of length = 1?

- Remember: BWT matrix rows = sorted suffixes of T
  - All suffixes prefixed by P occur in a consecutive set of rows
  - Possible to calculate the first and last position
  - Occurrences = Last - First +1
- How do we search?
  - Use C
  - Find *First* and *Last*

Backward-search algorithm

How do we find a pattern of length = 1?

- Remember: BWT matrix rows = sorted suffixes of T
  - All suffixes prefixed by P occur in a consecutive set of rows
  - Possible to calculate the first and last position
  - Occurrences = Last - First +1
- How do we search?
  - Use C

i	m	p	s
1	5	6	8

  - Find *First* and *Last*

F		L
#	mississipp	i
i	#mississip	p
i	ppi#missis	s
i	ssippi#mis	s
i	ssissippi#	m
m	ississippi	#
p	i#mississi	p
p	pi#mississ	i
s	ippi#missi	s
s	issippi#mi	s
s	sippi#miss	i
s	sissippi#m	i

Backward-search algorithm

How do we find a pattern of length = 1?

- Remember: BWT matrix rows = sorted suffixes of T
  - All suffixes prefixed by P occur in a consecutive set of rows
  - Possible to calculate the first and last position
  - Occurrences = Last - First +1
- How do we search?
  - Use C

i	m	p	s
1	5	6	8

  - Find *First* and *Last*

F		L
#	mississipp	i
i	#mississip	p
i	ppi#missis	s
i	ssippi#mis	s
i	ssissippi#	m
m	ississippi	#
p	i#mississi	p
p	pi#mississ	i
s	ippi#missi	s
s	issippi#mi	s
s	sippi#miss	i
s	sissippi#m	i

Backward-search algorithm

How do we find a pattern **ssi**?

- Find *First* and *Last* of P[j]
- Determine L[*First*, *Last*]
- Find the first P[j-1]in L[*First*, *Last*]
  Find the last P[j-1]in L[*First*, *Last*]
- Use L-to-F mapping
- j = j-1 $\rightarrow$ Repetition

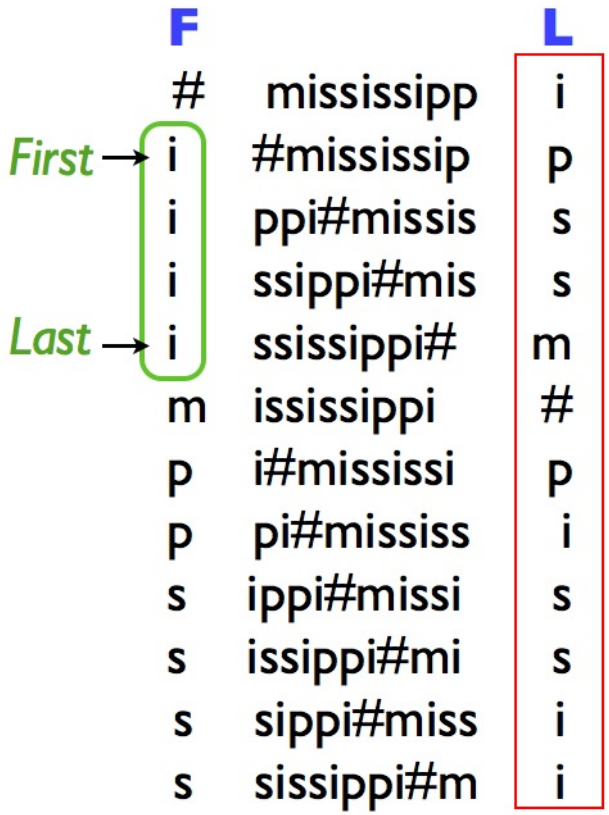
- Termination by finding **ssi**
- Occurrence = 12-11+1 = 2

Backward-search algorithm

How do we find a pattern **ssi**?

- Find *First* and *Last* of P[j]
- Determine L[*First*, *Last*]
- Find the first P[j-1]in L[*First*, *Last*]
  Find the last P[j-1]in L[*First*, *Last*]
- Use L-to-F mapping
- j = j-1 $\rightarrow$ Repetition

- Termination by finding **ssi**
- Occurrence = 12-11+1 = 2

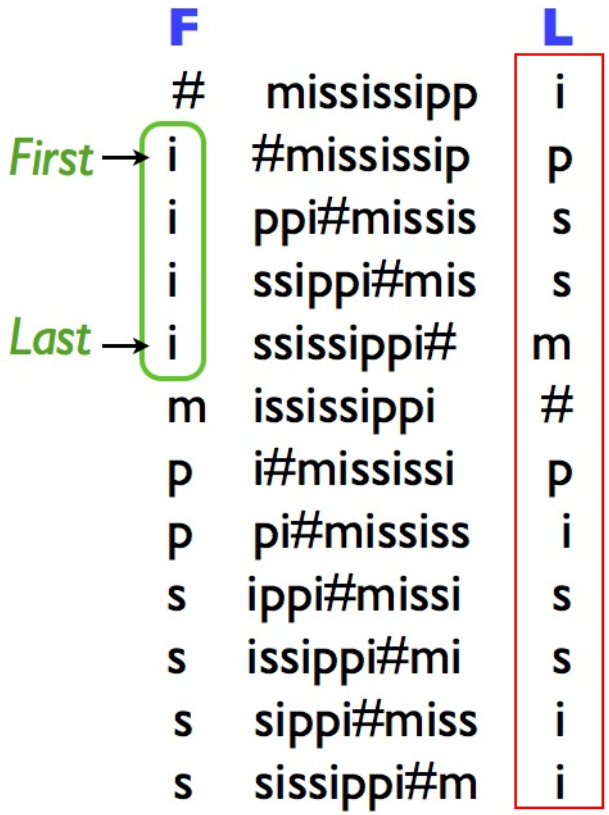


Backward-search algorithm

How do we find a pattern **ssi**?

- Find *First* and *Last* of P[j]
- Determine L[*First*, *Last*]
- Find the first P[j-1]in L[*First*, *Last*]
  Find the last P[j-1]in L[*First*, *Last*]
- Use L-to-F mapping
- j = j-1 $\rightarrow$ Repetition

- Termination by finding **ssi**
- Occurrence = 12-11+1 = 2

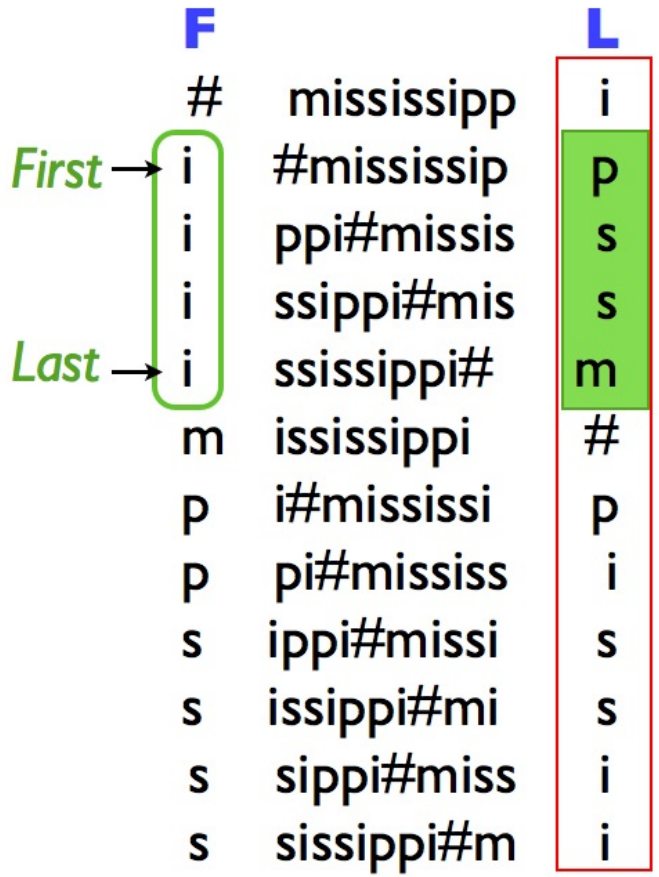


Backward-search algorithm

How do we find a pattern **ssi**?

- Find *First* and *Last* of P[j]
- Determine L[*First*, *Last*]
- Find the first P[j-1]in L[*First*, *Last*]
  Find the last P[j-1]in L[*First*, *Last*]
- Use L-to-F mapping
- j = j-1 $\rightarrow$ Repetition

- Termination by finding **ssi**
- Occurrence = 12-11+1 = 2

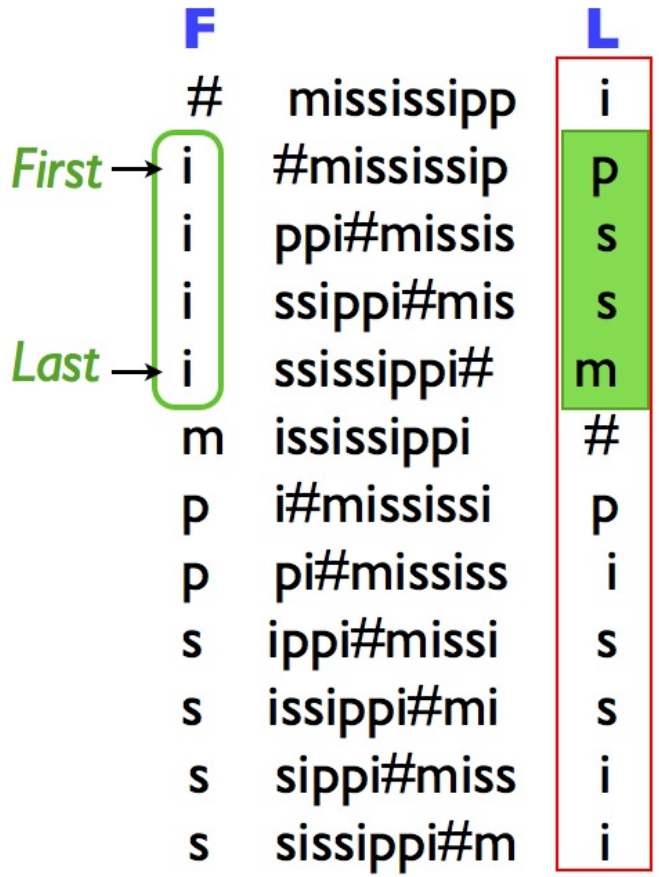


Backward-search algorithm

How do we find a pattern **ssi**?

- Find *First* and *Last* of P[j]
- Determine L[*First*, *Last*]
- Find the first P[j-1]in L[*First*, *Last*]
  Find the last P[j-1]in L[*First*, *Last*]
- Use L-to-F mapping
- j = j-1 $\rightarrow$ Repetition

- Termination by finding **ssi**
- Occurrence = 12-11+1 = 2

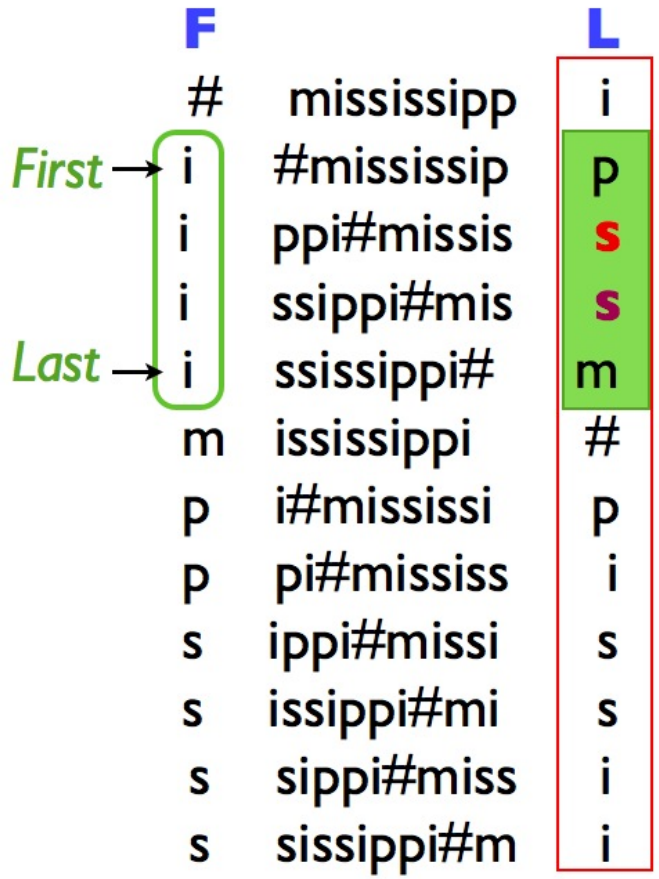


Backward-search algorithm

How do we find a pattern **ssi**?

- Find *First* and *Last* of P[j]
- Determine L[*First*, *Last*]
- Find the first P[j-1]in L[*First*, *Last*]
  Find the last P[j-1]in L[*First*, *Last*]
- Use L-to-F mapping
- j = j-1 $\rightarrow$ Repetition

- Termination by finding **ssi**
- Occurrence = 12-11+1 = 2

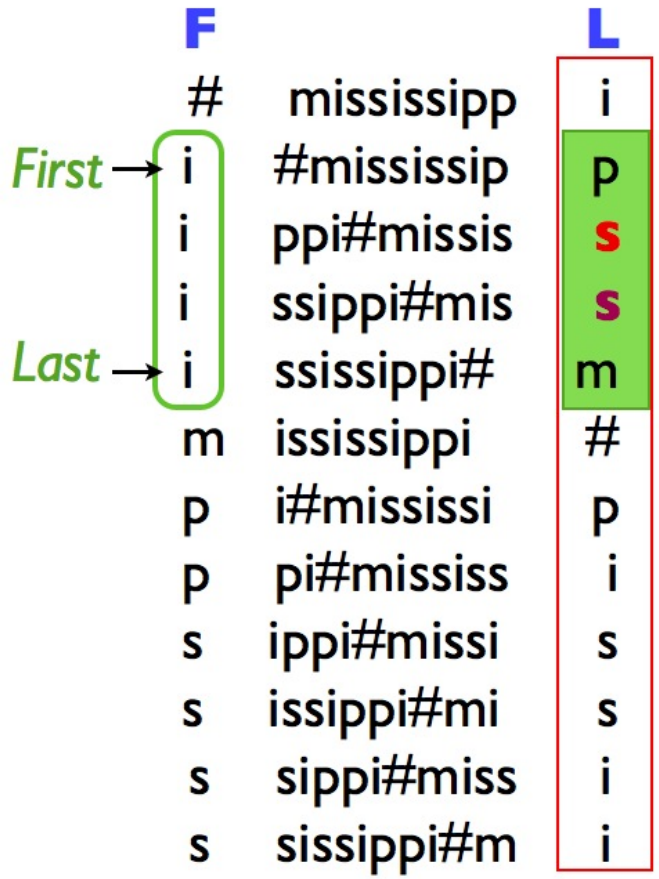


Backward-search algorithm

How do we find a pattern **ssi**?

- Find *First* and *Last* of P[j]
- Determine L[*First*, *Last*]
- Find the first P[j-1]in L[*First*, *Last*]
  Find the last P[j-1]in L[*First*, *Last*]
- Use L-to-F mapping
- j = j-1 $\rightarrow$ Repetition

- Termination by finding **ssi**
- Occurrence = 12-11+1 = 2

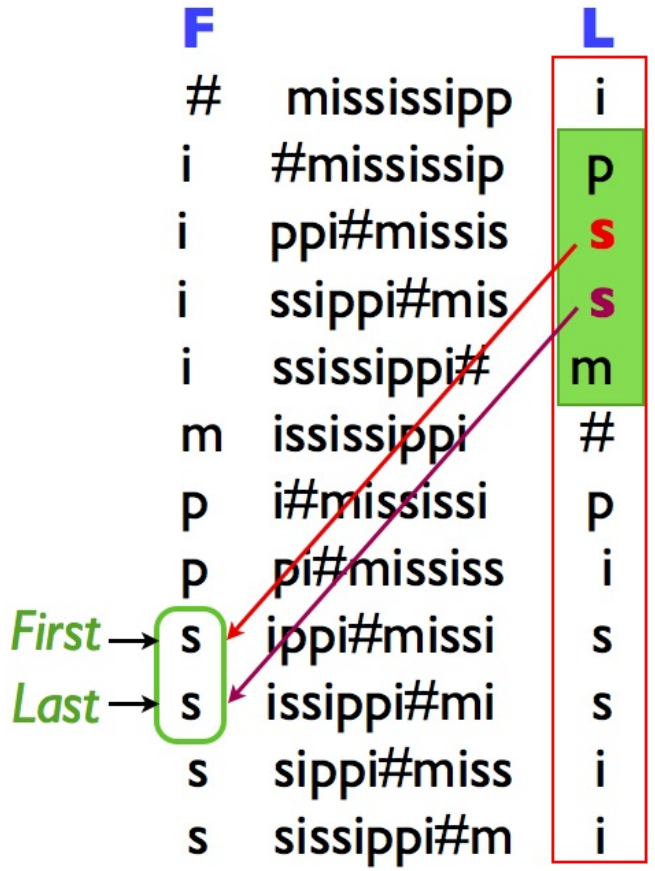


Backward-search algorithm

How do we find a pattern **ssi**?

- Find *First* and *Last* of P[j]
- Determine L[*First*, *Last*]
- Find the first P[j-1]in L[*First*, *Last*]
  Find the last P[j-1]in L[*First*, *Last*]
- Use L-to-F mapping
- j = j-1 $\rightarrow$ Repetition

- Termination by finding **ssi**
- Occurrence = 12-11+1 = 2

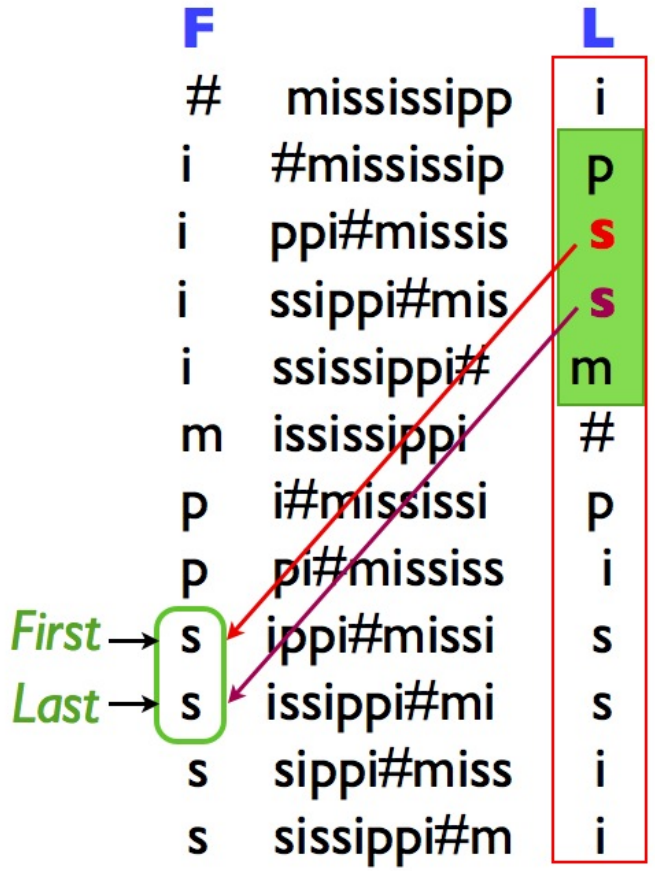


Backward-search algorithm

How do we find a pattern **ssi**?

- Find *First* and *Last* of P[j]
- Determine L[*First*, *Last*]
- Find the first P[j-1]in L[*First*, *Last*]
  Find the last P[j-1]in L[*First*, *Last*]
- Use L-to-F mapping
- j = j-1 $\rightarrow$ Repetition

- Termination by finding **ssi**
- Occurrence = 12-11+1 = 2

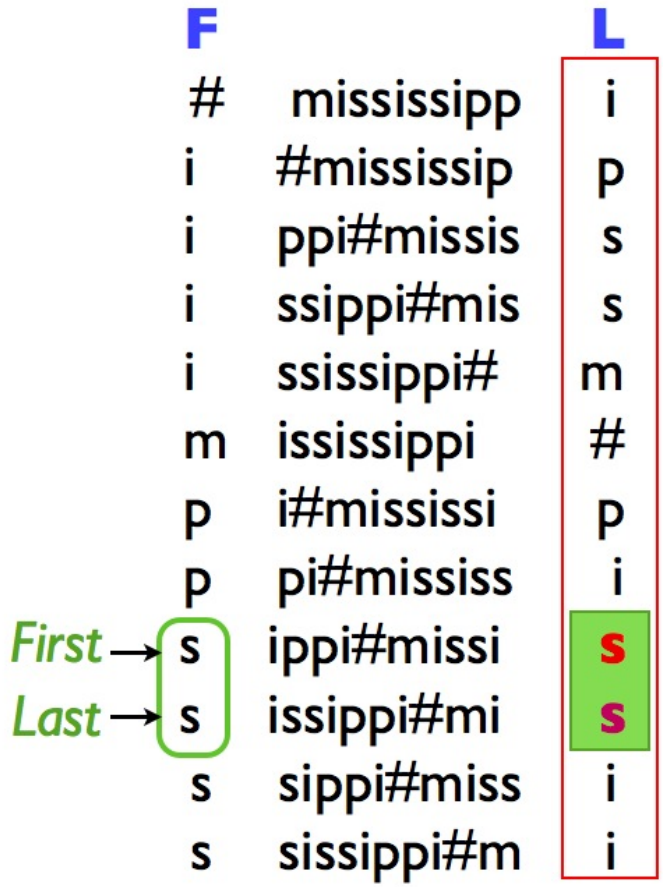


Backward-search algorithm

How do we find a pattern **ssi**?

- Find *First* and *Last* of P[j]
- Determine L[*First*, *Last*]
- Find the first P[j-1]in L[*First*, *Last*]
  Find the last P[j-1]in L[*First*, *Last*]
- Use L-to-F mapping
- j = j-1 $\rightarrow$ Repetition

- Termination by finding **ssi**
- Occurrence = 12-11+1 = 2

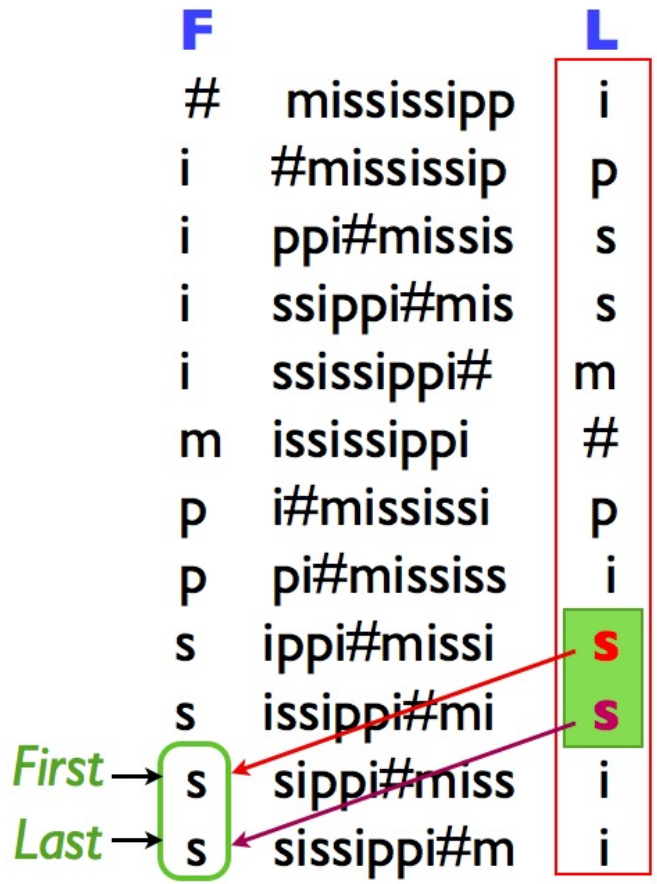


Backward-search algorithm

How do we find a pattern **ssi**?

- Find *First* and *Last* of P[j]
- Determine L[*First*, *Last*]
- Find the first P[j-1]in L[*First*, *Last*]
  Find the last P[j-1]in L[*First*, *Last*]
- Use L-to-F mapping
- j = j-1 $\rightarrow$ Repetition

- Termination by finding **ssi**
- Occurrence = 12-11+1 = 2

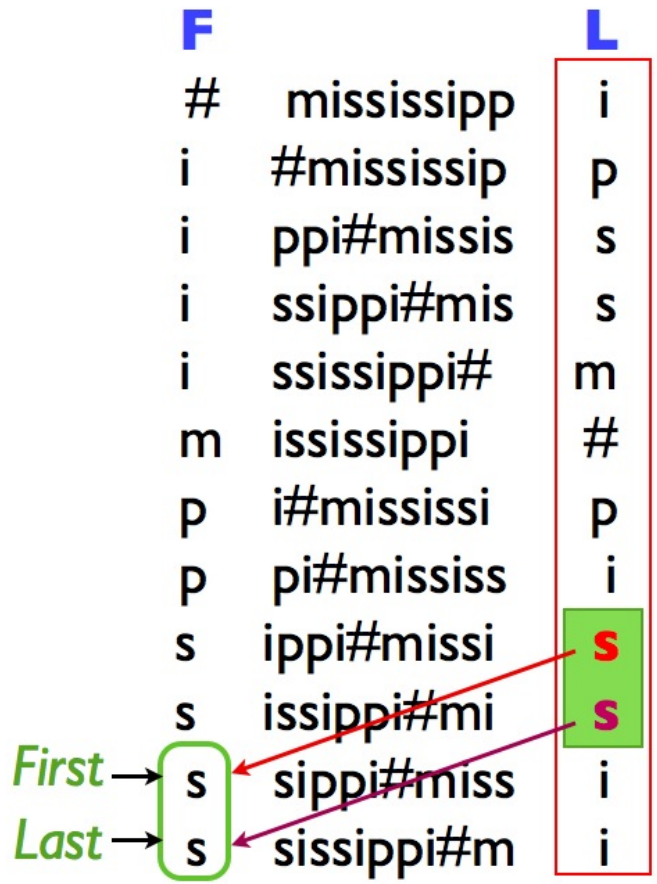


Backward-search algorithm

How do we find a pattern **ssi**?

- Find *First* and *Last* of P[j]
- Determine L[*First*, *Last*]
- Find the first P[j-1]in L[*First*, *Last*]
  Find the last P[j-1]in L[*First*, *Last*]
- Use L-to-F mapping
- j = j-1 $\rightarrow$ Repetition

- Termination by finding **ssi**
- Occurrence = 12-11+1 = 2

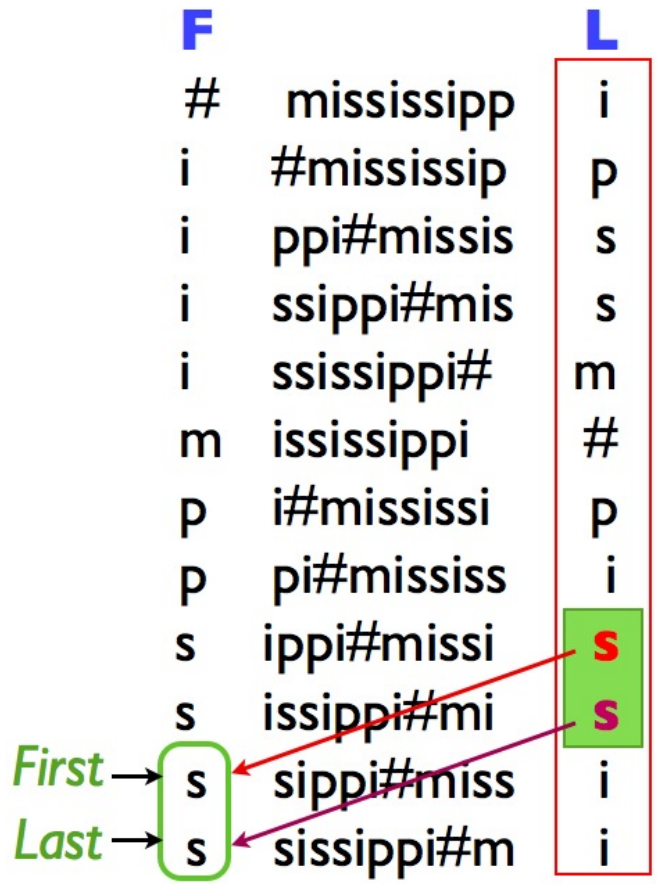


Backward-search algorithm

How do we find a pattern **ssi**?

- Find *First* and *Last* of P[j]
- Determine L[*First*, *Last*]
- Find the first P[j-1]in L[*First*, *Last*]
  Find the last P[j-1]in L[*First*, *Last*]
- Use L-to-F mapping
- j = j-1 $\rightarrow$ Repetition

- Termination by finding **ssi**
- Occurrence = 12-11+1 = 2



Backward-search algorithm

Algorithm backward_search(P[1,p])

(1) i←p, c←P[p], **First**←C[c]+1, **Last**←C[c+1]
(2) **while** ((First≤Last) **and** (i≥ 2)) **do**
  (3) c←P[i-1];
  (4) **First**←C[c]+ Occ(c,**First**-1)+1;
  (5) **Last**←C[c]+ Occ(c,**Last**);
  (6) i←i-1;
(7) **if** (Last<First)
  then return „no rows prefixed by P[1,p]“
  else return ⟨First, Last⟩

- Running time: O(p)

Outline

Algorithm to locate P in T

- Every i= *First, First+1, ..., Last* presents the pattern
- Possible because all characters have the same order in L and F
- Locate the position in T for every i

m	i	s	s	i	s	s	i	p	p	i	#
1	2	3	4	5	6	7	8	9	10	11	12

	F		L	
	#	mississipp	i	1
	i	#mississip	p	2
	i	ppi#missis	s	3
	i	ssippi#mis	s	4
	i	ssissippi#	m	5
	m	ississippi	#	6
	p	i#mississi	p	7
	p	pi#mississ	i	8
First →	s	ippi#missi	s	9
Last →	s	issippi#mi	s	10
	s	sippi#miss	i	11
	s	sissippi#m	i	12

Algorithm to locate P in T

Backward-step

- Locate the position $pos_T(i)$ in T for every i

m	i	s	s	i	s	s	i	p	p	i	#
1	2	3	4	5	6	7	8	9	10	11	12

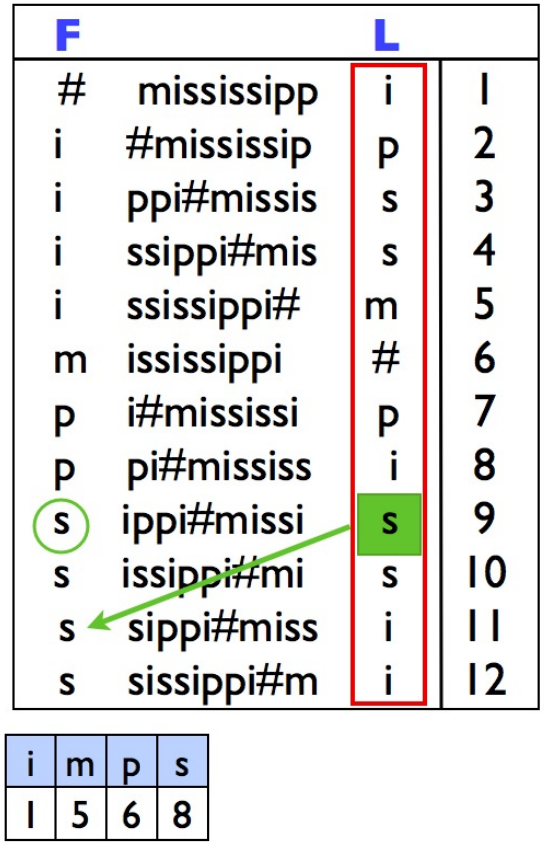
- Problem: Can't find pos_T(i) in T directly
  But: It's possible to find it indirectly
- Given index j and it's position in T
  $\rightarrow$ Can find pos_T(i) such that
  $pos_T(i) = pos_T(j) - 1$
- Backward_step algorithm

	F		L	
	#	mississipp	i	1
	i	#mississip	p	2
	i	ppi#missis	s	3
	i	ssippi#mis	s	4
	i	ssissippi#	m	5
	m	ississippi	#	6
	p	i#mississi	p	7
	p	pi#mississ	i	8
First →	s	ippi#missi	s	9
Last →	s	issippi#mi	s	10
	s	sippi#miss	i	11
	s	sissippi#m	i	12

Algorithm to locate P in T

Backward-step

- How does it work?
  - *You have:* j and $pos_T(j)$, a compressed L
  - *You want:* index i of L of T[$pos_T(j)$-1]
  - Determine L[j]:
    Compare Occ(c, j) with Occ(c,j-i) for every c $\in \Sigma \cup \{\#\}$
  - Use L-to-F mapping:
    pos(i)=C[L[j]] + Occ(L[j], i)
- Calling Occ() is O(1)
  $\rightarrow$ Backward_step in O(1)



Algorithm to locate P in T

Preprocessing

- Mark every x-th character from T and its corresponding position in L
- Store them in S
- Example: pos(r_3) = 8

	F		L	
	#	mississipp	i	1
	i	#mississip	p	2
	i	ppi#missis	s	3
	i	ssippi#mis	s	4
	i	ssissippi#	m	5
	m	ississippi	#	6
	p	i#mississi	p	7
	p	pi#mississ	i	8
First →	s	ippi#missi	s	9
Last →	s	issippi#mi	s	10
	s	sippi#miss	i	11
	s	sissippi#m	i	12

m	i	s	s	i	s	s	i	p	p	i	#
1	2	3	4	5	6	7	8	9	10	11	12

Algorithm to locate P in T

- Find position of row i:
  - **If r_i is marked**, then return pos(r_i) from S
  - **Otherwise** use Backward_step(i)
    - Repeat t times until you find a marked row j
    - $pos_T(i) = pos_T(j) + t$

F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
First → s	ippi#missi	s	9
Last → s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

m	i	s	s	i	s	s	i	p	p	i	#
1	2	3	4	5	6	7	8	9	10	11	12

Algorithm to locate P in T

Finding **si**

- Determine all positions for i = First, First+1, ..., Last
- First = 9, Last = 10
- i= 10
  - r_{10} is marked
  - pos(10) = 4

m	1
i	2
s	3
s	4
i	5
s	6
s	7
i	8
p	9
p	10
i	11
#	12

	F	L	
	#	mississipp	i	1
	i	#mississip	p	2
	i	ppi#missis	s	3
	i	ssippi#mis	s	4
	i	ssissippi#	m	5
	m	ississippi	#	6
	p	i#mississi	p	7
	p	pi#mississ	i	8
	s	ippi#missi	s	9
i →	s	issippi#mi	s	10
	s	sippi#miss	i	11
	s	sissippi#m	i	12

Algorithm to locate P in T

Finding **si**

- Determine all positions for i = First, First+1, ..., Last
- First = 9, Last = 10
- i= 10
  - r_{10} is marked
  - pos(10) = 4

m	1
i	2
s	3
s	4
i	5
s	6
s	7
i	8
p	9
p	10
i	11
#	12

	F	L	
	#	mississipp	i	1
	i	#mississip	p	2
	i	ppi#missis	s	3
	i	ssippi#mis	s	4
	i	ssissippi#	m	5
	m	ississippi	#	6
	p	i#mississi	p	7
	p	pi#mississ	i	8
	s	ippi#missi	s	9
i →	s	issippi#mi	s	10
	s	sippi#miss	i	11
	s	sissippi#m	i	12

Algorithm to locate P in T

Finding **si**

- Determine all positions for i = First, First+1, ..., Last
- First = 9, Last = 10
- i= 10
  - r_{10} is marked
  - pos(10) = 4

m	1
i	2
s	3
s	4
i	5
s	6
s	7
i	8
p	9
p	10
i	11
#	12

	F	L	
	#	mississipp	i	1
	i	#mississip	p	2
	i	ppi#missis	s	3
	i	ssippi#mis	s	4
	i	ssissippi#	m	5
	m	ississippi	#	6
	p	i#mississi	p	7
	p	pi#mississ	i	8
	s	ippi#missi	s	9
i →	s	issippi#mi	s	10
	s	sippi#miss	i	11
	s	sissippi#m	i	12

Algorithm to locate P in T

Finding **si**

- i = 9
  - r_9 is not marked
  - Backward_step(9), t=1
  - r_{11} is not marked
  - Backward_step(11), t=2
  - r_4 is not marked
  - Backward_step(4), t=3
  - r_{10} is marked
  - pos(9) = pos(10) +3 = 4 + 3 = 7

m	1
i	2
s	3
s	4
i	5
s	6
s	7
i	8
p	9
p	10
i	11
#	12

	F	L	
	#	mississipp	i	1
	i	#mississip	p	2
	i	ppi#missis	s	3
	i	ssippi#mis	s	4
	i	ssissippi#	m	5
	m	ississippi	#	6
	p	i#mississi	p	7
	p	pi#mississ	i	8
i →	s	ippi#missi	s	9
	s	issippi#mi	s	10
	s	sippi#miss	i	11
	s	sissippi#m	i	12

Algorithm to locate P in T

Finding **si**

- i = 9
  - r_9 is not marked
  - Backward_step(9), t=1
  - r_{11} is not marked
  - Backward_step(11), t=2
  - r_4 is not marked
  - Backward_step(4), t=3
  - r_{10} is marked
  - pos(9) = pos(10) +3 = 4 + 3 = 7

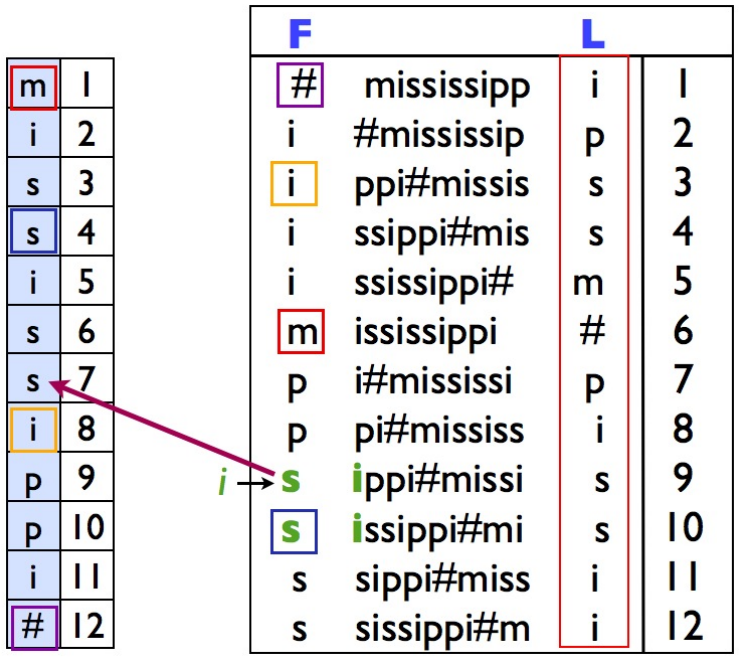
m	1
i	2
s	3
s	4
i	5
s	6
s	7
i	8
p	9
p	10
i	11
#	12

	F	L	
	# mississipp	i	1
	i #mississip	p	2
	i ppi#missis	s	3
	i ssippi#mis	s	4
	i ssissippi#	m	5
	m ississippi	#	6
	p i#mississi	p	7
	p pi#mississ	i	8
i →	s ippi#missi	s	9
	s issippi#mi	s	10
	s sippi#miss	i	11
	s sissippi#m	i	12

Algorithm to locate P in T

Finding **si**

- i = 9
  - r_9 is not marked
  - Backward_step(9), t=1
  - r_{11} is not marked
  - Backward_step(11), t=2
  - r_4 is not marked
  - Backward_step(4), t=3
  - r_{10} is marked
  - pos(9) = pos(10) +3 = 4 + 3 = 7



Algorithm to locate P in T

Finding **si**

- i = 9
  - r_9 is not marked
  - Backward_step(9), t=1
  - r_{11} is not marked
  - Backward_step(11), t=2
  - r_4 is not marked
  - Backward_step(4), t=3
  - r_{10} is marked
  - pos(9) = pos(10) +3 = 4 + 3 = 7

m	1
i	2
s	3
s	4
i	5
s	6
s	7
i	8
p	9
p	10
i	11
#	12

	F	L	
	# mississipp	i	1
	i #mississip	p	2
	i ppi#missis	s	3
	i ssippi#mis	s	4
	i ssissippi#	m	5
	m ississippi	#	6
	p i#mississi	p	7
	p pi#mississ	i	8
i →	s ippi#missi	s	9
	s issippi#mi	s	10
	s sippi#miss	i	11
	s sissippi#m	i	12

Algorithm to locate P in T

Finding **si**

- i = 9
  - r_9 is not marked
  - Backward_step(9), t=1
  - r_{11} is not marked
  - Backward_step(11), t=2
  - r_4 is not marked
  - Backward_step(4), t=3
  - r_{10} is marked
  - pos(9) = pos(10) +3 = 4 + 3 = 7

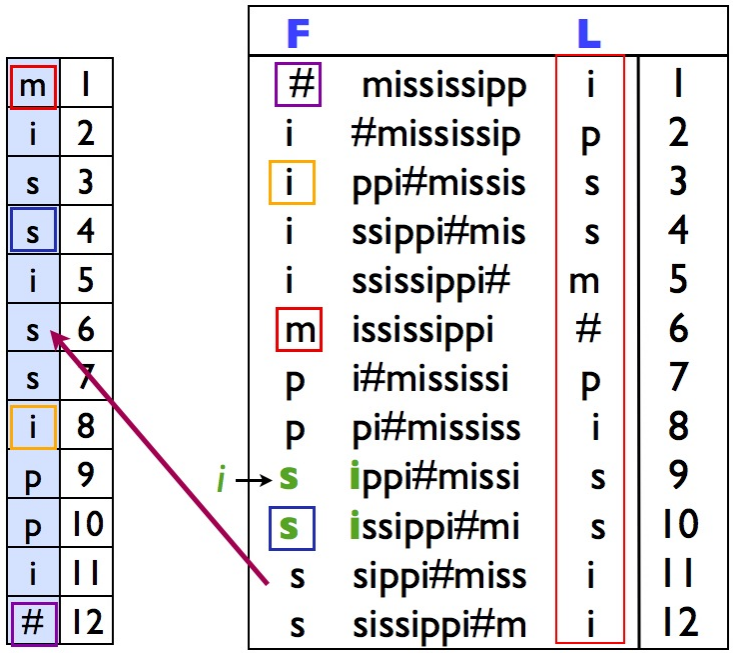
m	1
i	2
s	3
s	4
i	5
s	6
s	7
i	8
p	9
p	10
i	11
#	12

F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
i → s	ippi#missi	s	9
s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

Algorithm to locate P in T

Finding **si**

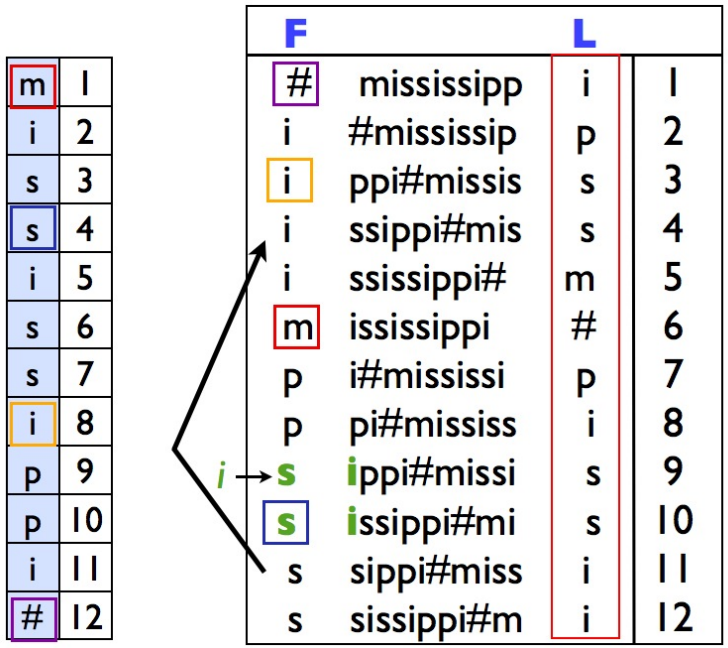
- i = 9
  - r_9 is not marked
  - Backward_step(9), t=1
  - r_{11} is not marked
  - Backward_step(11), t=2
  - r_4 is not marked
  - Backward_step(4), t=3
  - r_{10} is marked
  - pos(9) = pos(10) +3 = 4 + 3 = 7



Algorithm to locate P in T

Finding **si**

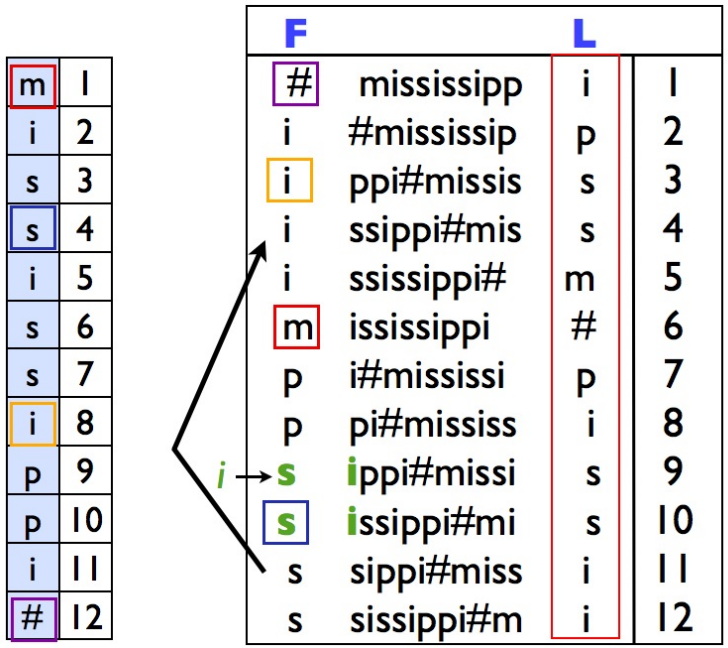
- i = 9
  - r_9 is not marked
  - Backward_step(9), t=1
  - r_{11} is not marked
  - Backward_step(11), t=2
  - r_4 is not marked
  - Backward_step(4), t=3
  - r_{10} is marked
  - pos(9) = pos(10) +3 = 4 + 3 = 7



Algorithm to locate P in T

Finding **si**

- i = 9
  - r_9 is not marked
  - Backward_step(9), t=1
  - r_{11} is not marked
  - Backward_step(11), t=2
  - r_4 is not marked
  - Backward_step(4), t=3
  - r_{10} is marked
  - pos(9) = pos(10) +3 = 4 + 3 = 7



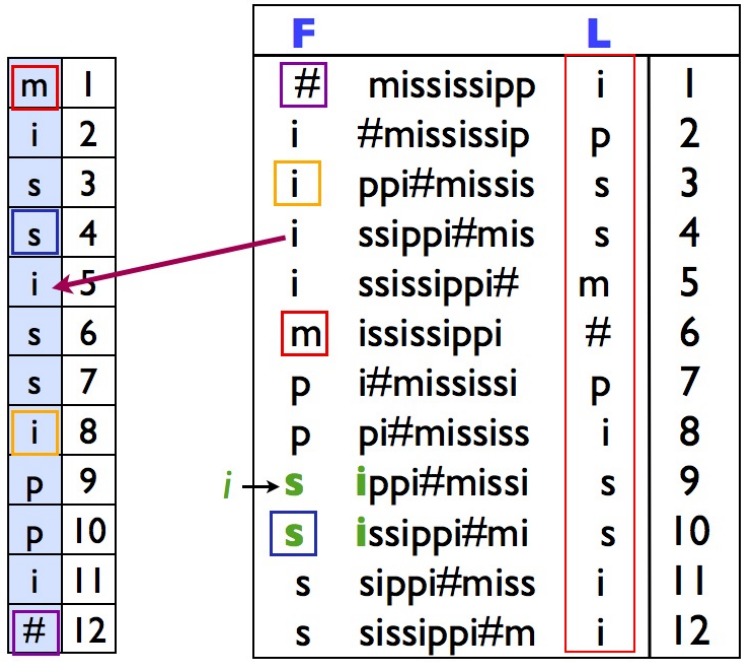
m	1
i	2
s	3
s	4
i	5
s	6
s	7
i	8
p	9
p	10
i	11
#	12

F		L	
#	mississipp	i	1
i	#mississip	p	2
i	ppi#missis	s	3
i	ssippi#mis	s	4
i	ssissippi#	m	5
m	ississippi	#	6
p	i#mississi	p	7
p	pi#mississ	i	8
s	ippi#missi	s	9
s	issippi#mi	s	10
s	sippi#miss	i	11
s	sissippi#m	i	12

Algorithm to locate P in T

Finding **si**

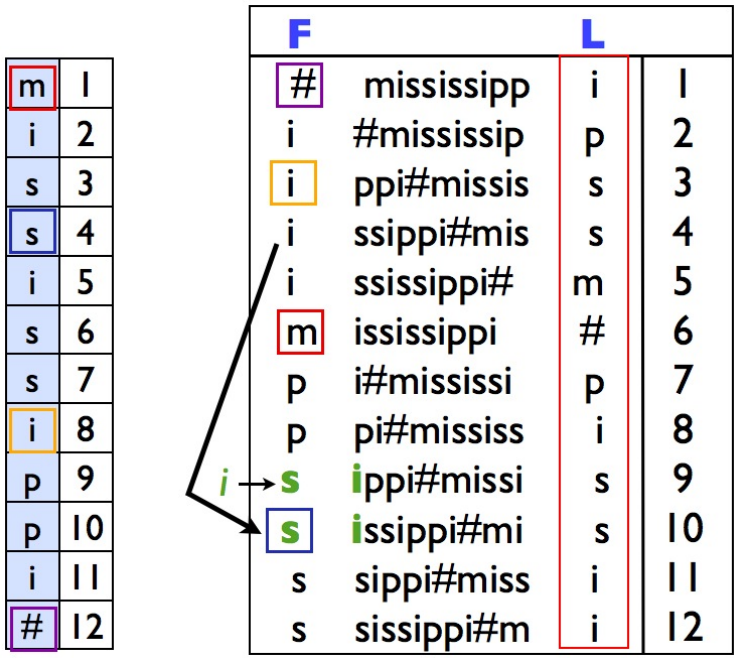
- i = 9
  - r_9 is not marked
  - Backward_step(9), t=1
  - r_{11} is not marked
  - Backward_step(11), t=2
  - r_4 is not marked
  - Backward_step(4), t=3
  - r_{10} is marked
  - pos(9) = pos(10) +3 = 4 + 3 = 7



Algorithm to locate P in T

Finding **si**

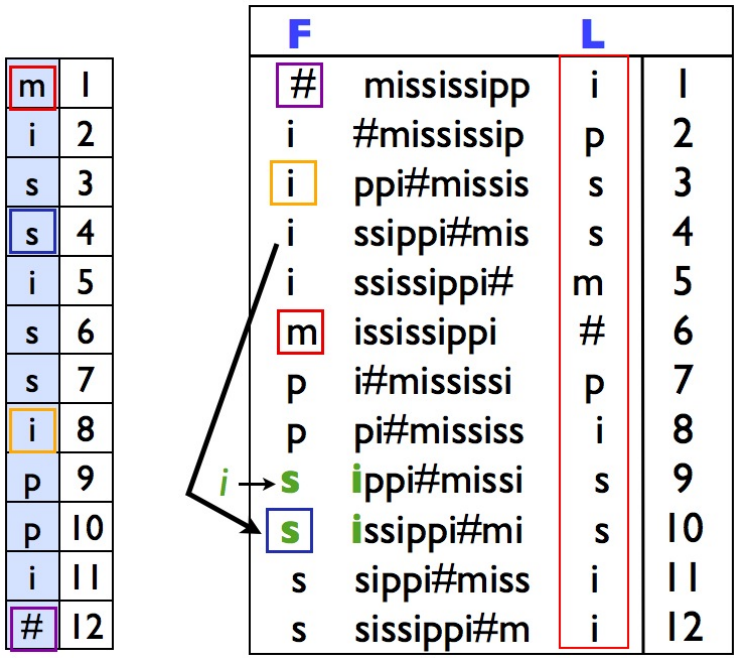
- i = 9
  - r_9 is not marked
  - Backward_step(9), t=1
  - r_{11} is not marked
  - Backward_step(11), t=2
  - r_4 is not marked
  - Backward_step(4), t=3
  - r_{10} is marked
  - pos(9) = pos(10) +3 = 4 + 3 = 7



Algorithm to locate P in T

Finding **si**

- i = 9
  - r_9 is not marked
  - Backward_step(9), t=1
  - r_{11} is not marked
  - Backward_step(11), t=2
  - r_4 is not marked
  - Backward_step(4), t=3
  - r_{10} is marked
  - pos(9) = pos(10) +3 = 4 + 3 = 7

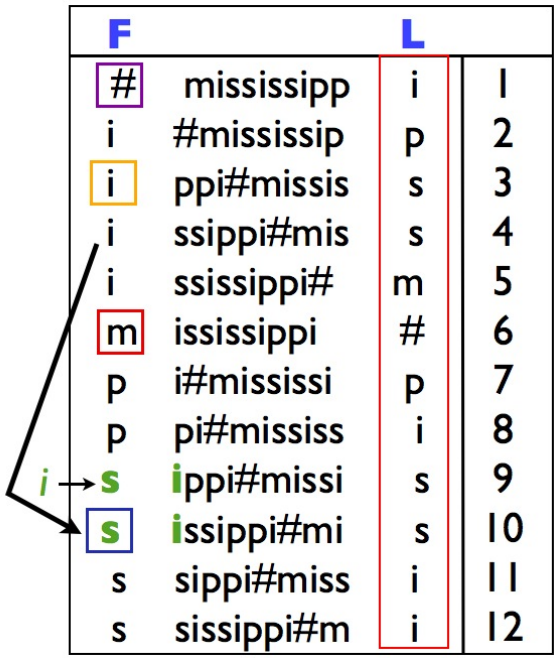


Algorithm to locate P in T

Finding **si**

- i = 9
  - r_9 is not marked
  - Backward_step(9), t=1
  - r_{11} is not marked
  - Backward_step(11), t=2
  - r_4 is not marked
  - Backward_step(4), t=3
  - r_{10} is marked
  - pos(9) = pos(10) +3 = 4 + 3 = 7

m	1
i	2
s	3
s	4
i	5
s	6
s	7
i	8
p	9
p	10
i	11
#	12



Outline

Mismatches

Due to:

- sequencing errors
- differences between reference and query organisms

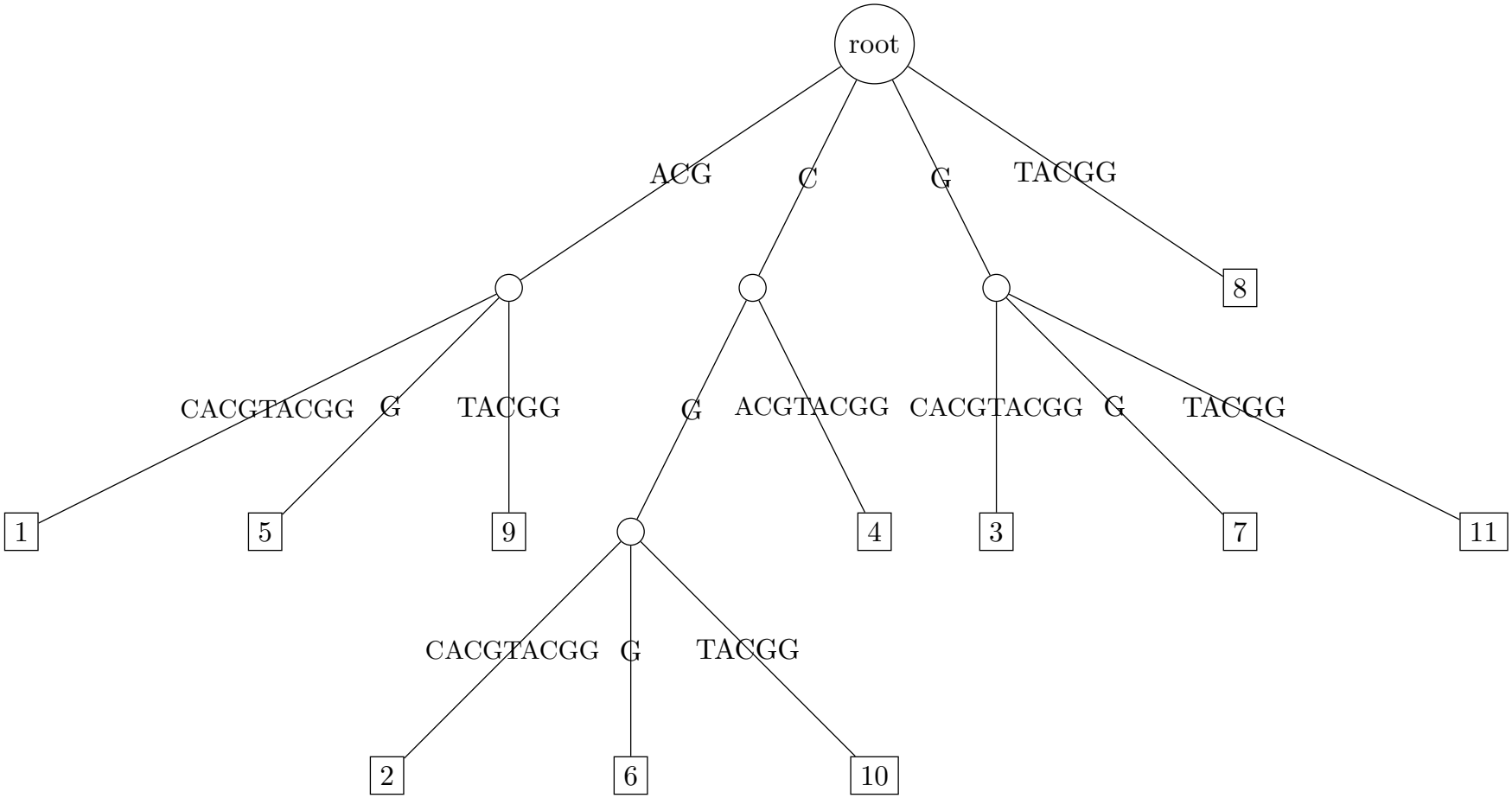
EXACTMATCH insufficient

Backtracking algorithm

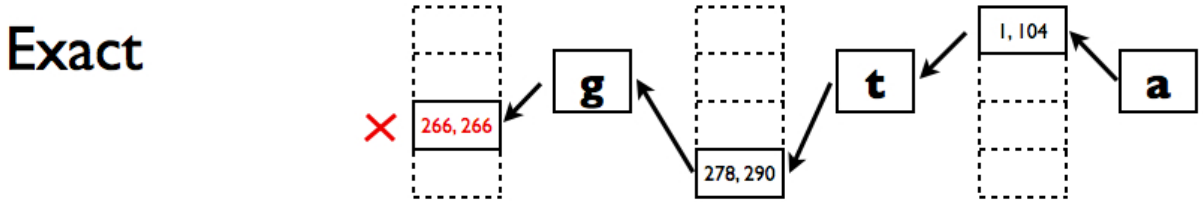
- Greedy
- Depth-First Search
- Allowed mismatches $\uparrow \Rightarrow$ runtime $\uparrow$

Example

ACGCACGTACGG

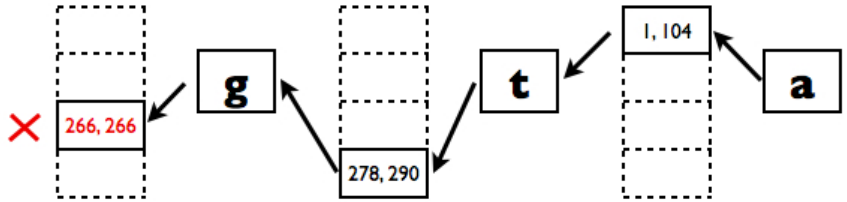


ggta

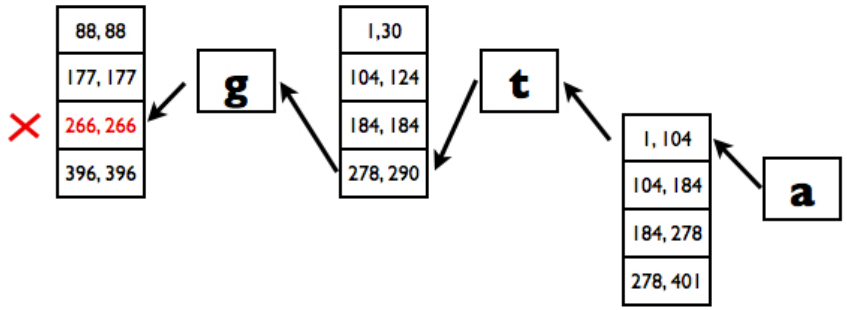


ggta

Exact

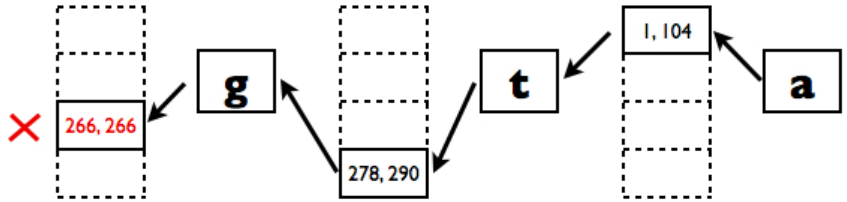


Inexact

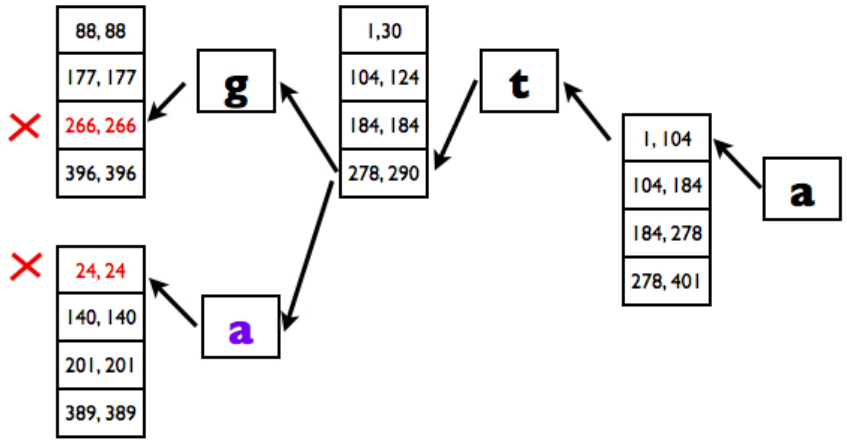


ggta

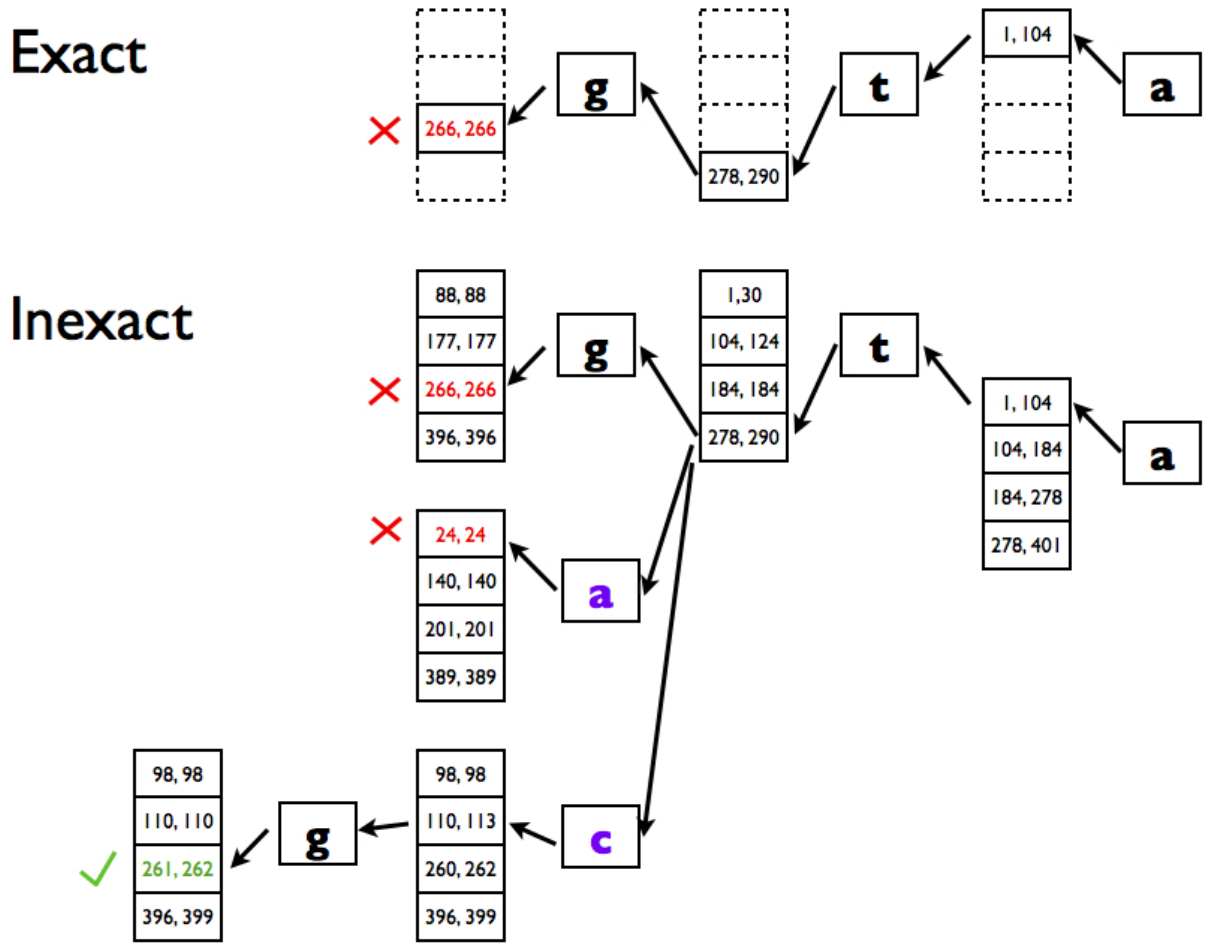
Exact



Inexact



ggta


Exact
266, 266
g
278, 290
t
1, 104
a
Inexact
88, 88
177, 177
266, 266
396, 396
g
1,30
104, 124
184, 184
278, 290
t
1, 104
104, 184
184, 278
278, 401
a
24, 24
140, 140
201, 201
389, 389
a
98, 98
110, 110
261, 262
396, 399
g
98, 98
110, 113
260, 262
396, 399
c

Outline

Excessive backtracking

- Avoid excessive backtracking
- Solution $\rightarrow$ double indexing
- and backtracking ceiling

Sources


Ben Langmead, Cole Trapnell, Mihai Pop & Steven L Salzberg.
Ultrafast and memory-efficient alignment of short DNA sequences to the human genome.
Genome Biology 2009, 10:R25.


M. Burrows & D.J. Wheeler
A block-sorting lossless data compression algorithm.
Digital SRC Research Report 1994.

Thank you

Questions?