EECS730: Introduction to Bioinformatics

Lecture 16: Next-generation sequencing

http://blog.illumina.com/images/default-source/Blog/next-generation-sequencing.jpg?sfvrsn=0

Some slides were adapted from Dr. Shaojie Zhang (University of Central Florida), Karl Kingsford (Carnegie Mellon University), and Ben Langmead (John Hopkins University)

Why sequencing

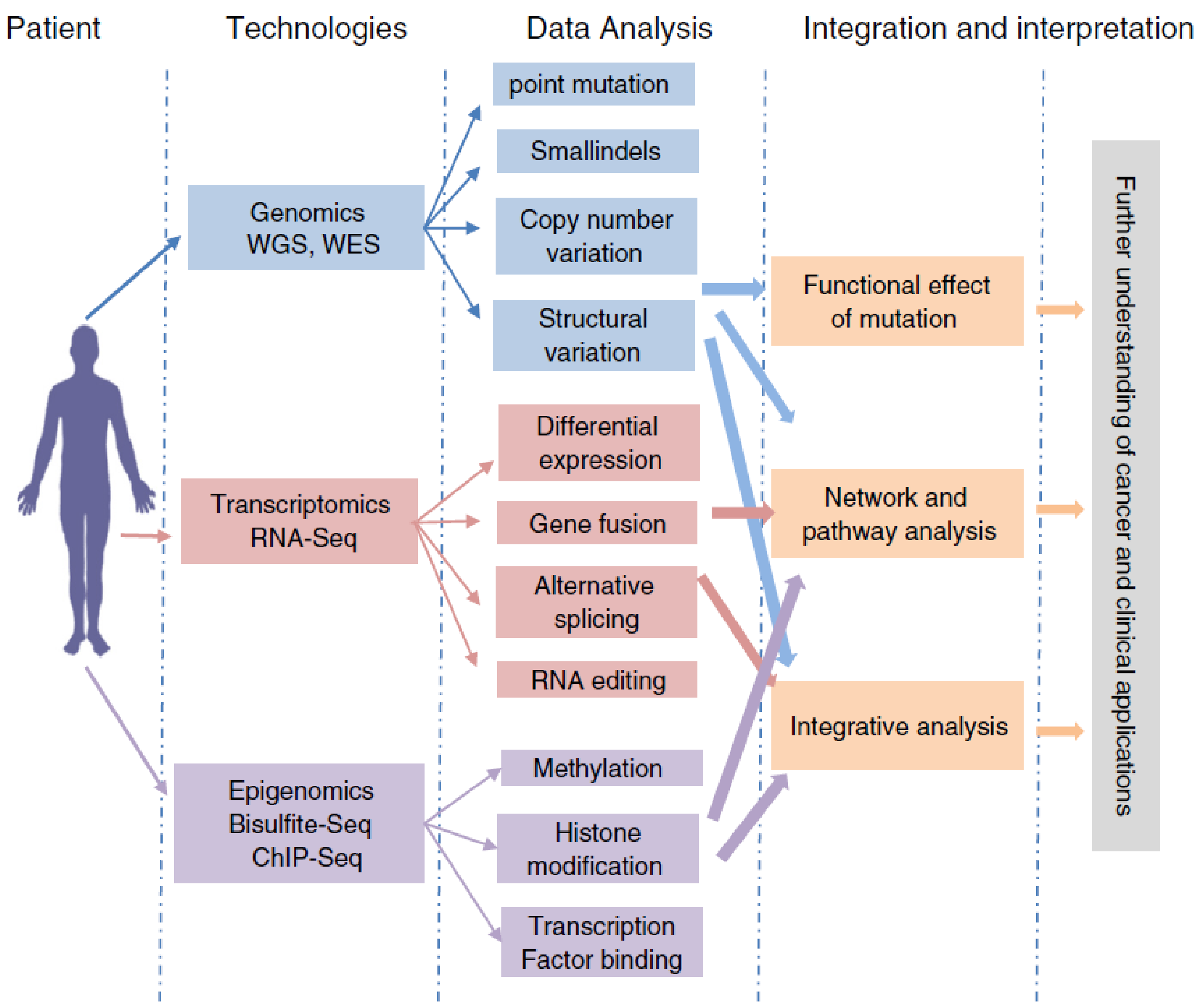
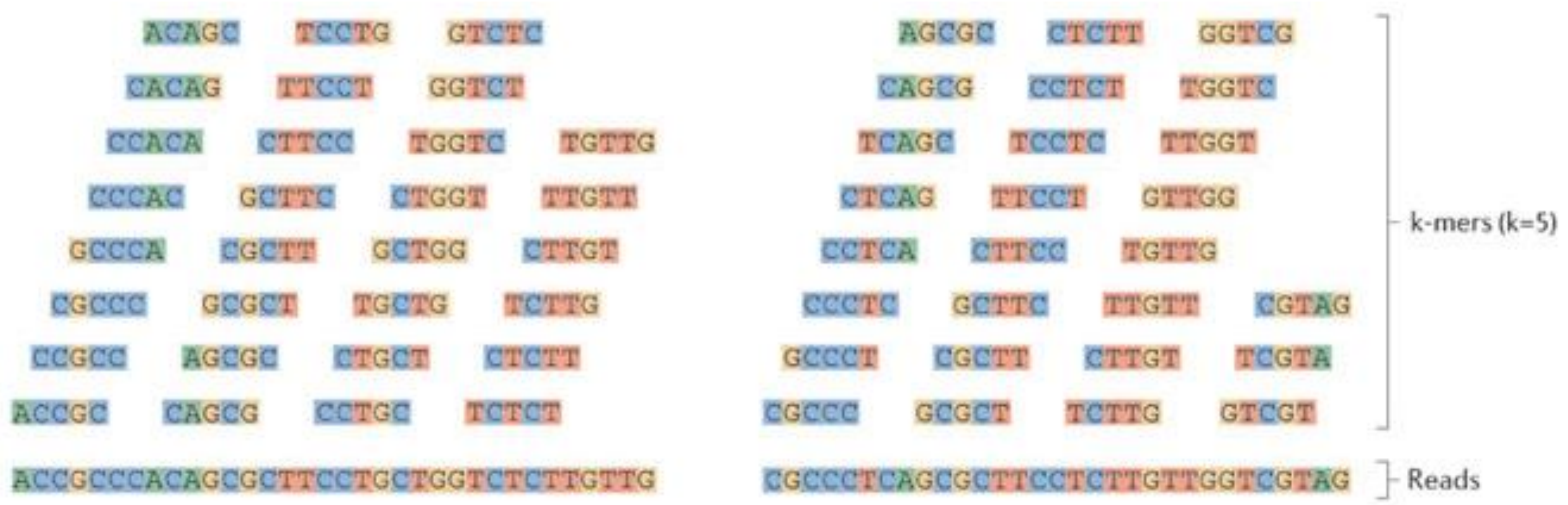


Figure 1 The workflow of integrating omics data in cancer research and clinical application. NGS technologies detect the genomic, transcriptomic and epigenomic alternations including mutations, copy number variations, structural variants, differentially expressed genes, fusion transcripts, DNA methylation change, etc. Various kinds of bioinformatics tools are used to analyze, integrate, and interpret the data to improve our understanding of cancer biology and develop personalized treatment strategy.

Shyr and Liu, 2013

Sequencing using microarray



http://homes.cs.washington.edu/~dcjones/assembly-primer/kmers.png

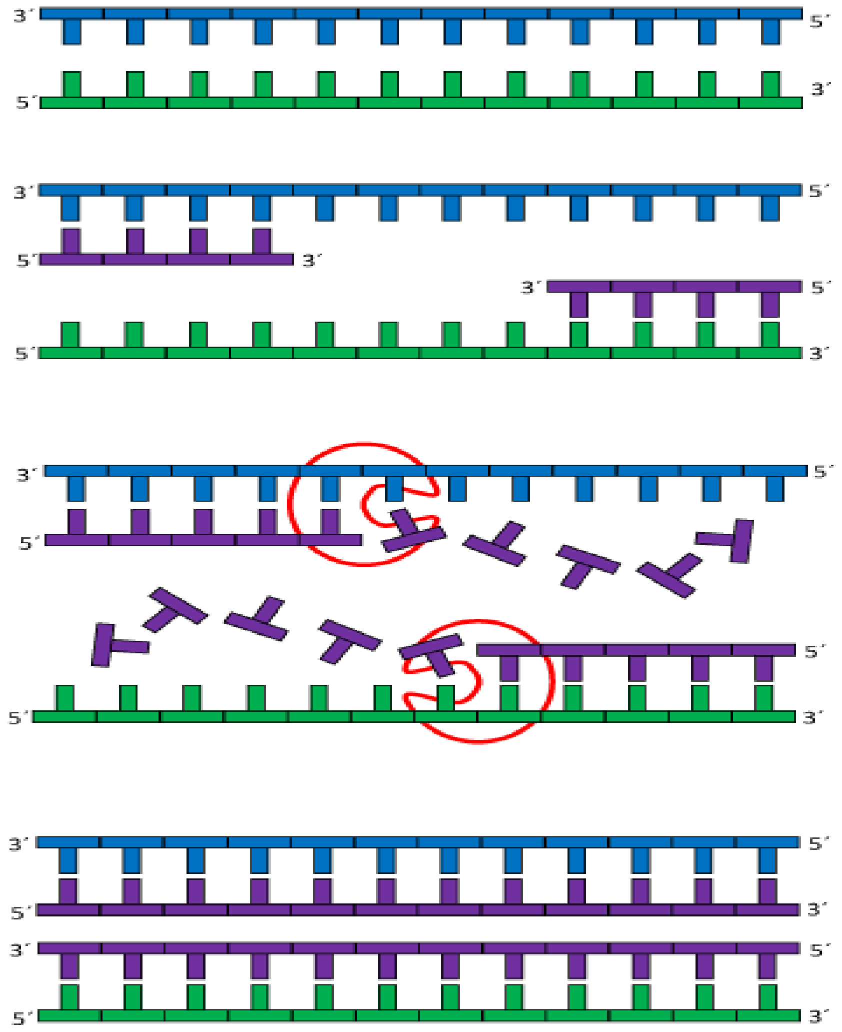
Sequencing advantages

- **Unbiased detection of novel transcripts:** Unlike arrays, sequencing technology does not require species- or transcript-specific probes. It can detect novel transcripts, gene fusions, single nucleotide variants, indels (small insertions and deletions), and other previously unknown changes that arrays cannot detect.
- **Broader dynamic range:** With array hybridization technology, gene expression measurement is limited by background at the low end and signal saturation at the high end. Sequencing technology quantifies discrete, digital sequencing read counts, offering a broader dynamic range.
- **Easier detection of rare and low-abundance transcripts:** Sequencing coverage depth can easily be increased to detect rare transcripts, single transcripts per cell, or weakly expressed genes.

http://www.illumina.com/technology/next-generation-sequencing/mrna-seq.html

Outline of sequencing mechanism

- Take the target DNA molecule as a template
- Utilize signals that are emitted when incorporating different nucleic acids
- Read out and parse the signal to determine the sequence of the DNA



Sanger sequencing

- Developed by Frederick Sanger (shared the 1980 Nobel Prize) and colleagues in 1977, it was the most widely used sequencing method for approximately 39 years.

- Gold standard for sequencing today

- Accurate (>99.99% accuracy) and produce long reads (>500bp)

- Relatively expensive ($2400 per 1Mbp) and low throughput

Sanger sequencing

- This method begins with the use of special enzymes to synthesize fragments of DNA that terminate when a selected base appears in the stretch of DNA being sequenced.
- These fragments are then sorted according to size by placing them in a slab of polymeric gel and applying an electric field -- a technique called electrophoresis.
- Because of DNA's negative charge, the fragments move across the gel toward the positive electrode. The shorter the fragment, the faster it moves.
- Typically, each of the terminating bases within the collection of fragments is tagged with a radioactive probe for identification.

An example

Problem Statement: Consider the following DNA sequence (from firefly luciferase). Draw the sequencing
gel pattern that forms as a result of sequencing the following template DNA with ddNTP as the capper.

atgaccatgattacg...

Solution:

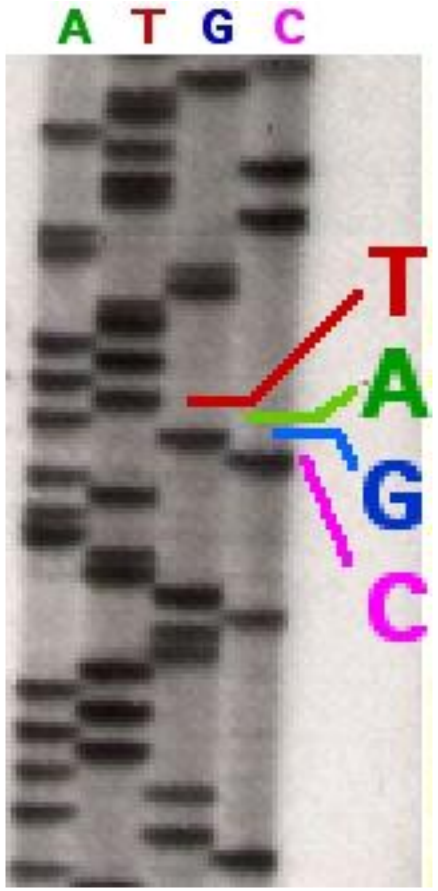
Given DNA template:	5'-atgaccatgattacg...-3'
DNA synthesized:	3'-tactggtactaatgc...-5'

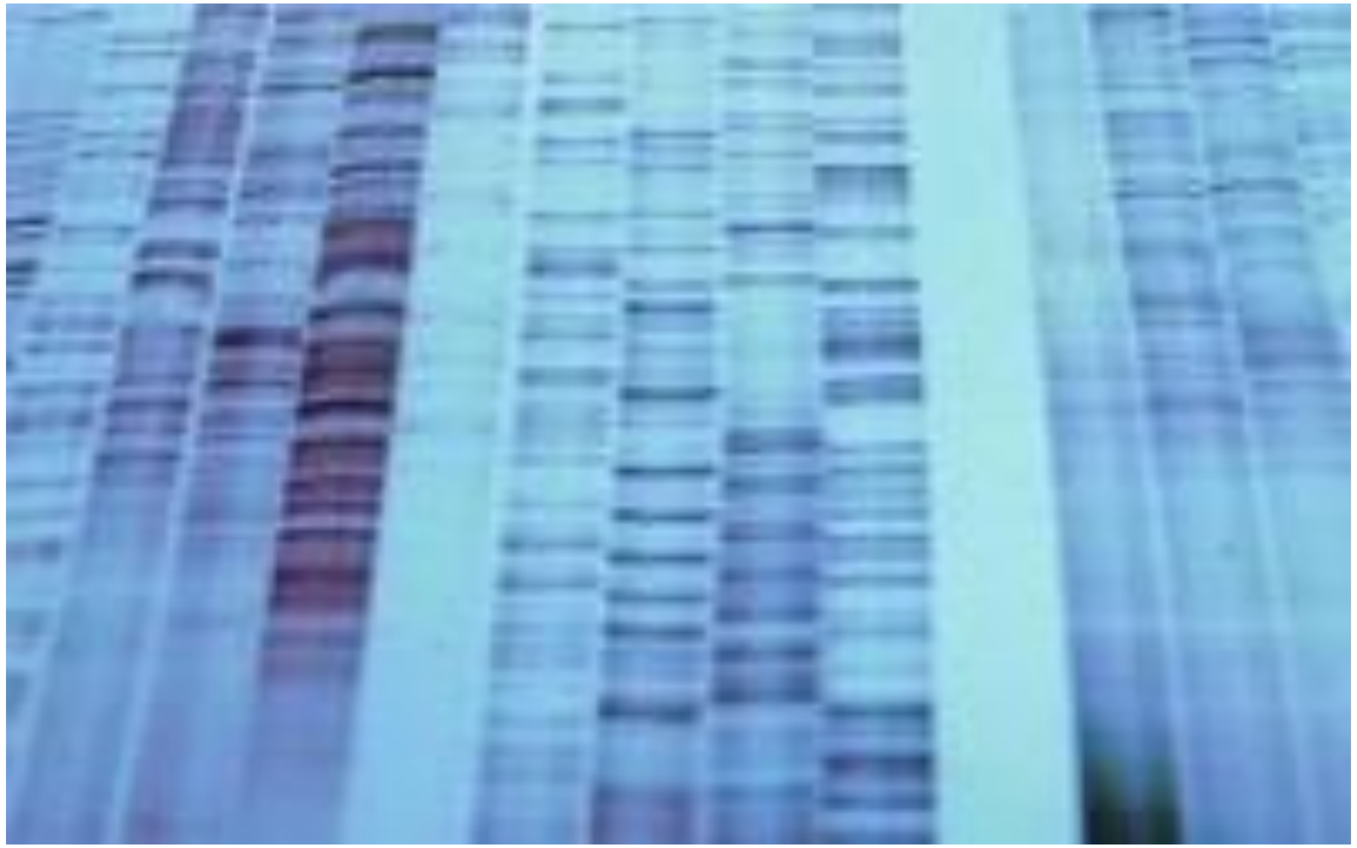
An example

```
Given DNA template:    5'-atgaccatgattacg...-3'
  DNA synthesized:      3'-tactggtactaatgc...-5'
Gel pattern:                     +--------------------------+
          lane ddATP              | W  |       |    ||          |
          lane ddTTP              | W |  |   |   |  |         |
          lane ddCTP              | W   |       |       |     |
          lane ddGTP              | W      ||          |      |
                                 +--------------------------+
                                     Electric Field        +
                                     Decreasing size
```

where "W" indicates the well position, and "|" denotes the DNA bands on the sequencing gel.

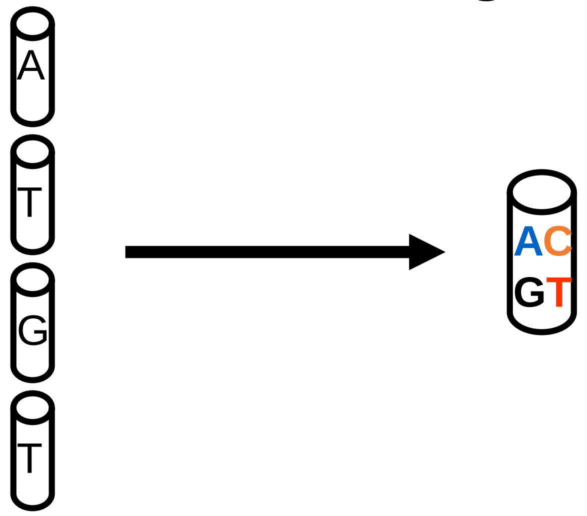
An example





Capillary electrophoresis

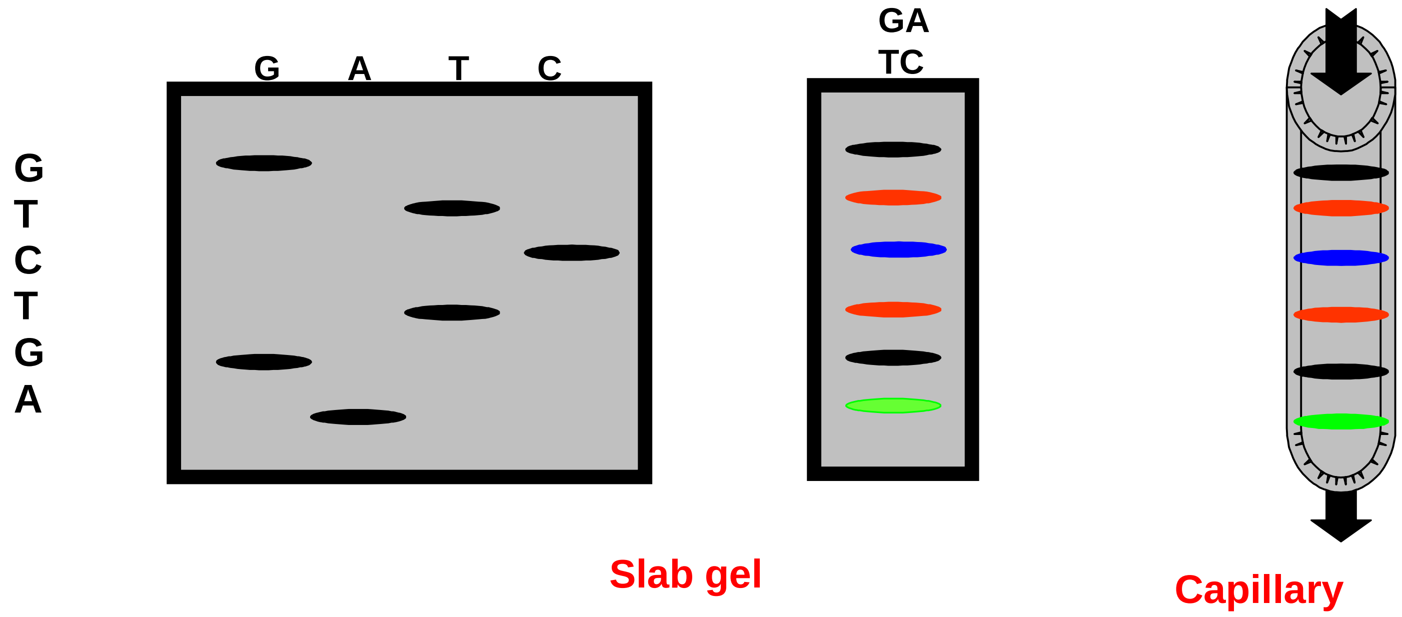
- A distinct dye or “color” is used for each of the four ddNTP.
- Since the terminating nucleotides can be distinguished by color, all four reactions can be performed in a single tube.



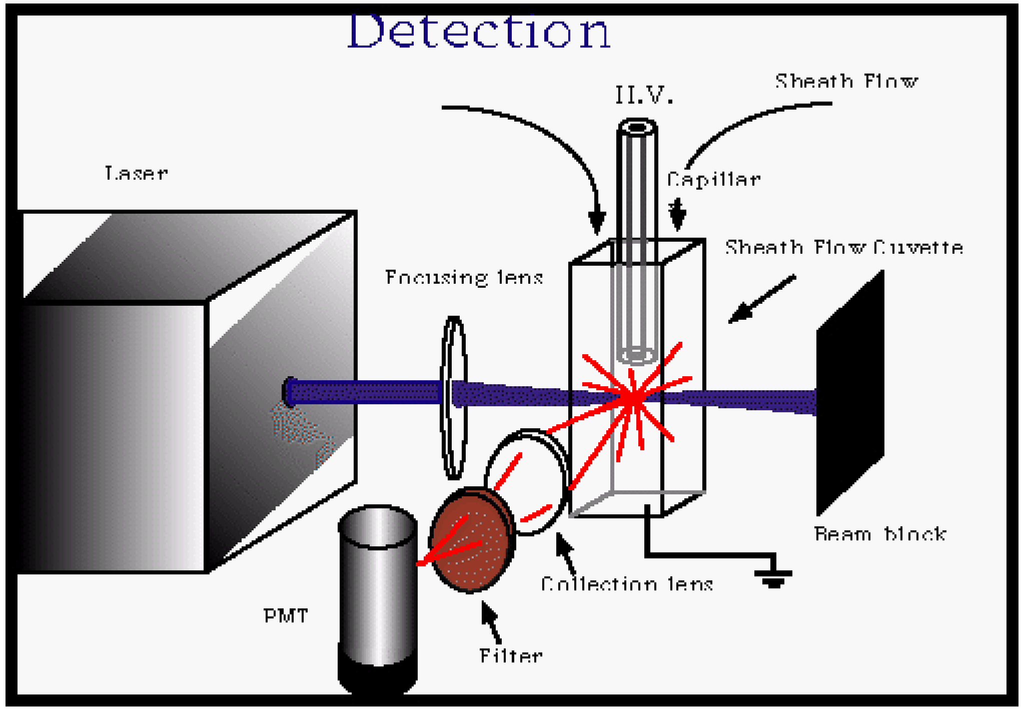
The fragments are distinguished by size and “color.”

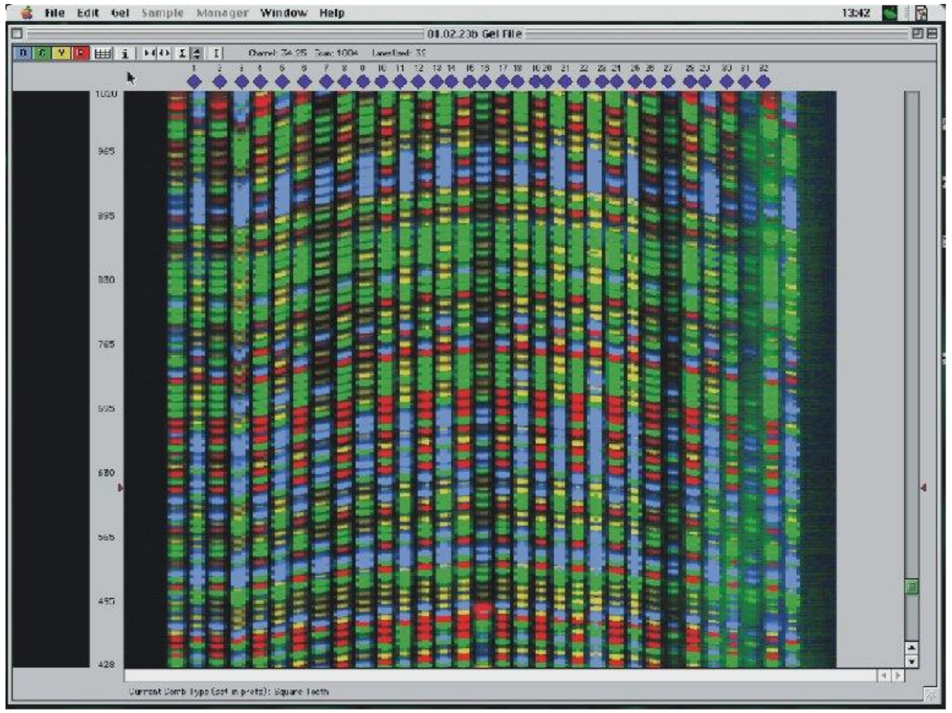
Capillary electrophoresis

The DNA ladder is resolved in one gel lane or in a capillary.



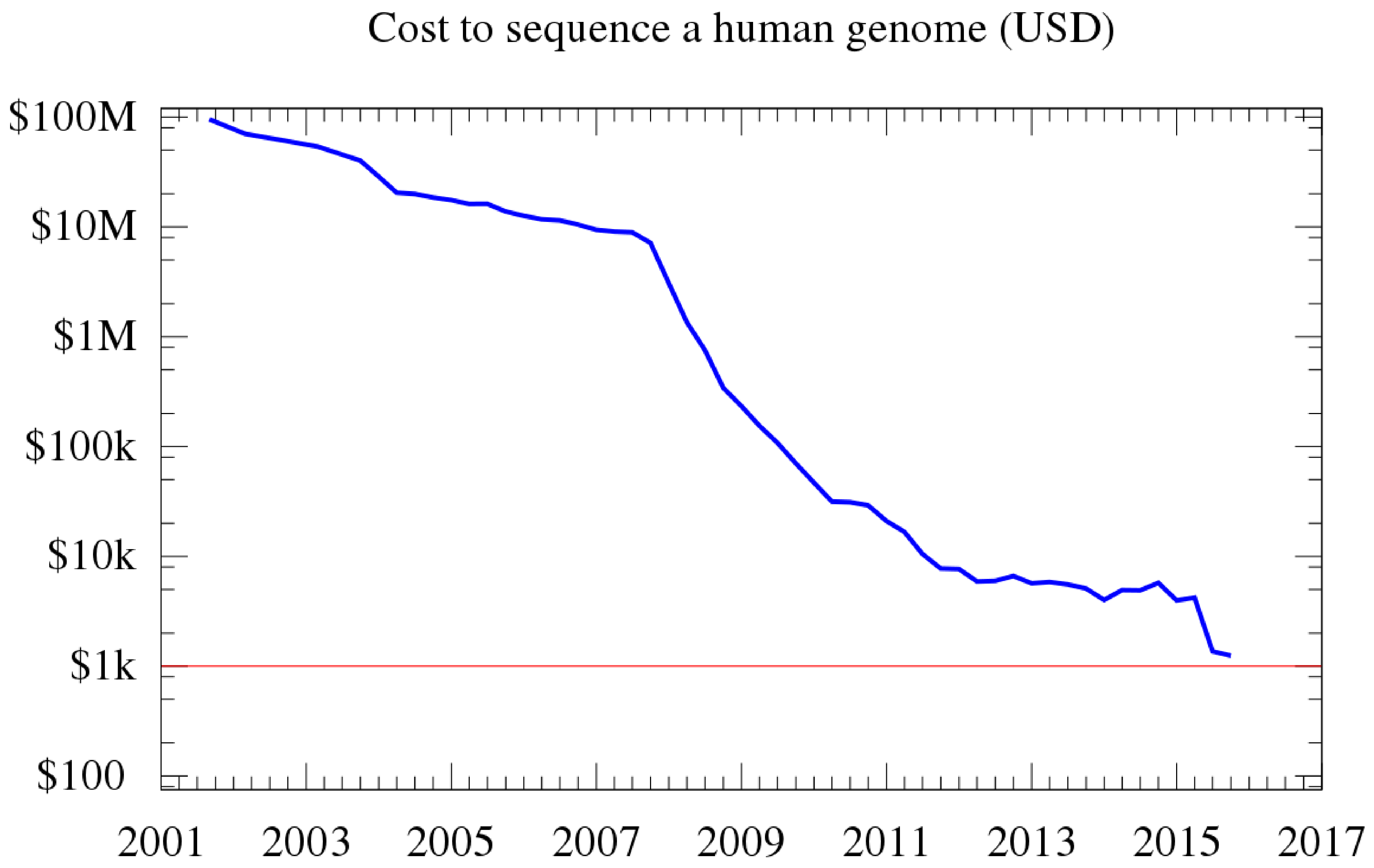
Capillary electrophoresis





Next-generation sequencing

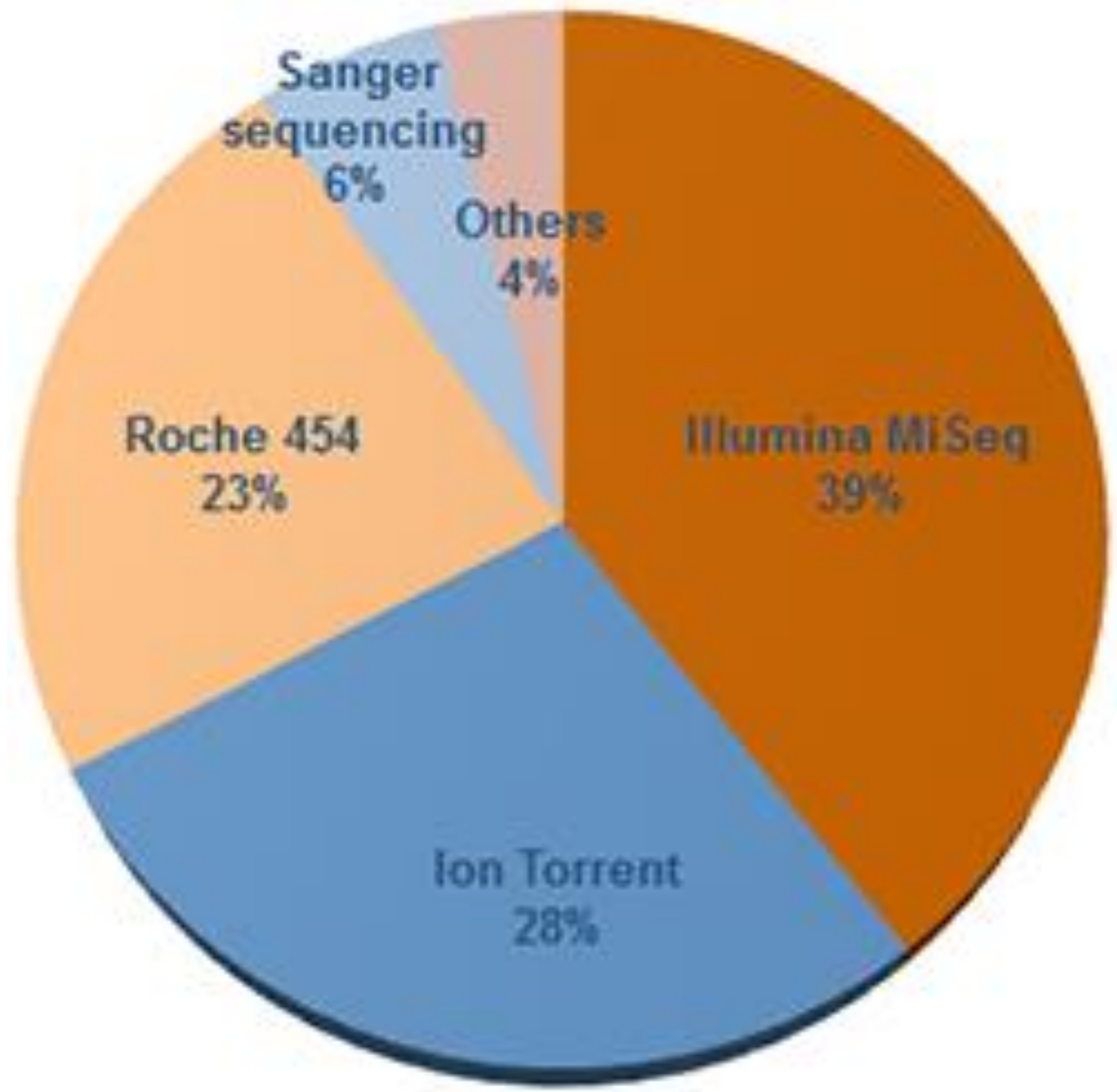
- First generation sequencing (Sanger sequencing)
- Next generation sequencing (current)
  - AKA:
    - Second generation sequencing
    - Massively parallel sequencing
    - Ultra high-throughput sequencing



https://en.wikipedia.org/wiki/Whole_genome_sequencing#/media/File:Historic_cost_of_sequencing_a_human_genome.svg

NGS platforms

- **Illumina/Solexa**
- ABI SOLiD
- Roche 454
- Polonator
- HeliScope
- …



http://ngs-expert.com/wp-content/uploads/2013/06/amplicon_sequencing.png

Comparison between platforms

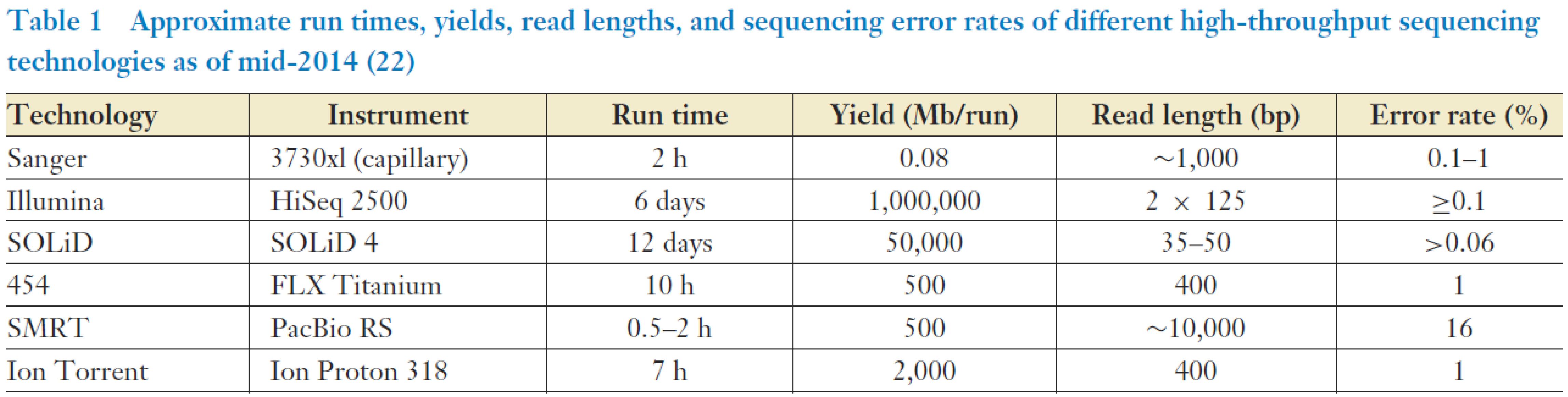


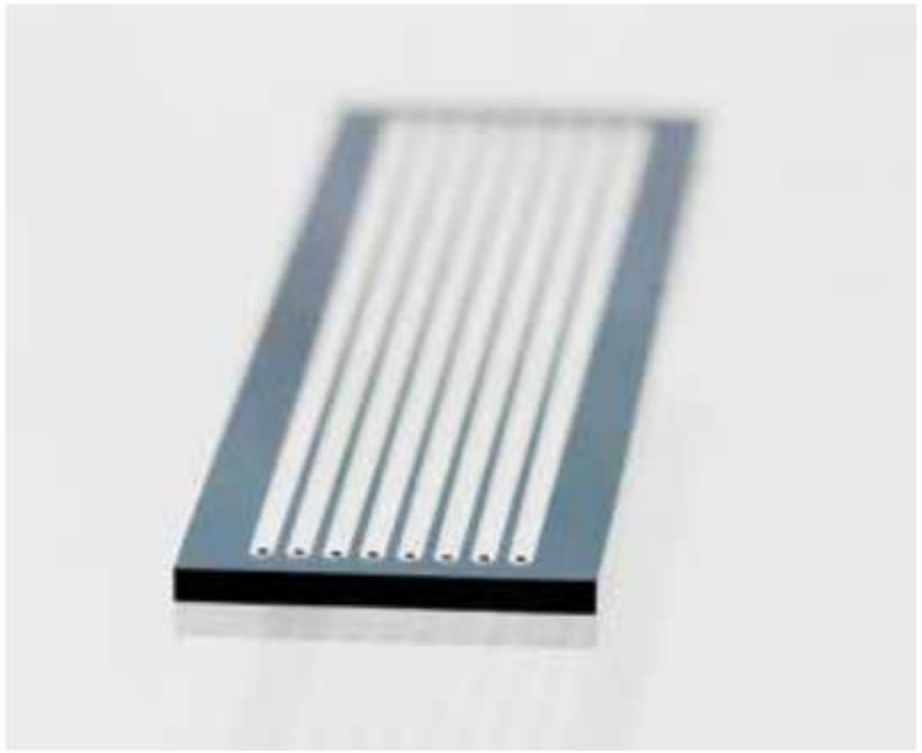
Table 1 Approximate run times, yields, read lengths, and sequencing error rates of different high-throughput sequencing technologies as of mid-2014 (22)

Technology	Instrument	Run time	Yield (Mb/run)	Read length (bp)	Error rate (%)
Sanger	3730xl (capillary)	2 h	0.08	~1,000	0.1–1
Illumina	HiSeq 2500	6 days	1,000,000	2 × 125	≥0.1
SOLiD	SOLiD 4	12 days	50,000	35–50	>0.06
454	FLX Titanium	10 h	500	400	1
SMRT	PacBio RS	0.5–2 h	500	~10,000	16
Ion Torrent	Ion Proton 318	7 h	2,000	400	1

Reinert et al., 2015, Annual Review Genomics & Human Genetics

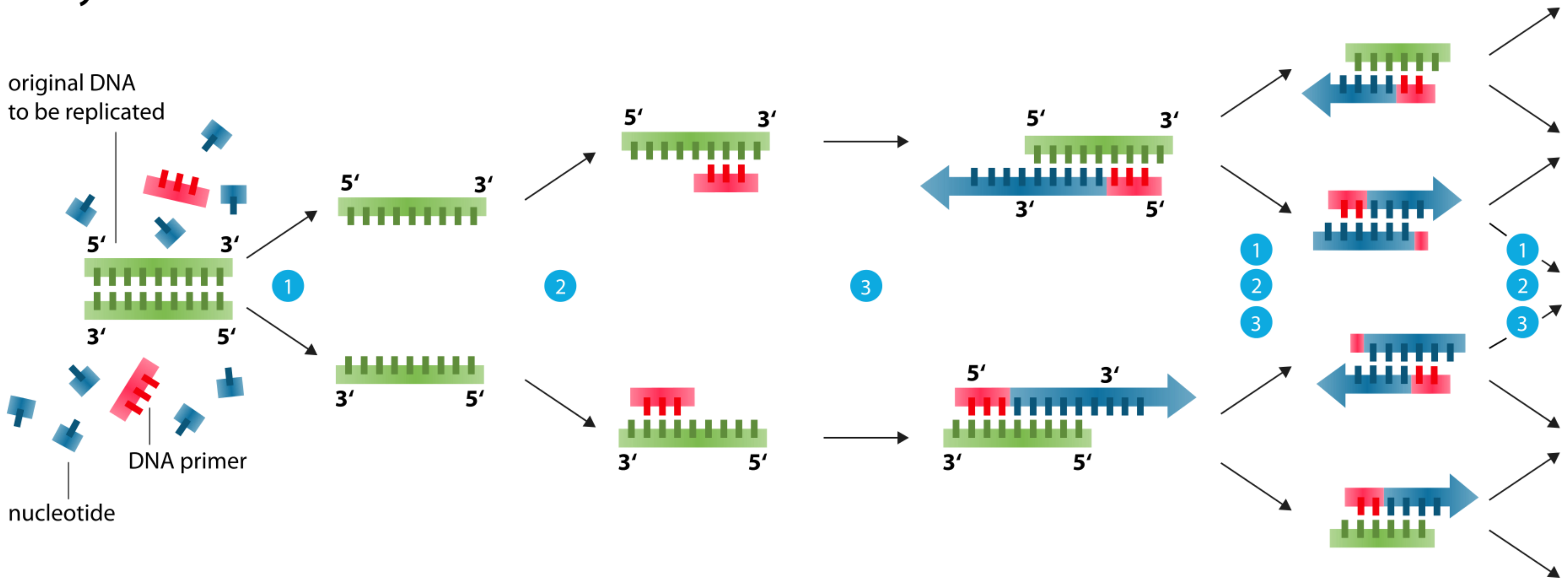
Illumina (Solexa) technology

- Also being performed on a glass slide
- Each spot on the slide correspond to a cluster of the same DNA molecule
- Library preparation
  - Fragment DNA and tag the fragments with adaptors
  - Use PCR to amplify the tagged DNA fragments



Illumina flow cell

Polymerase chain reaction - PCR



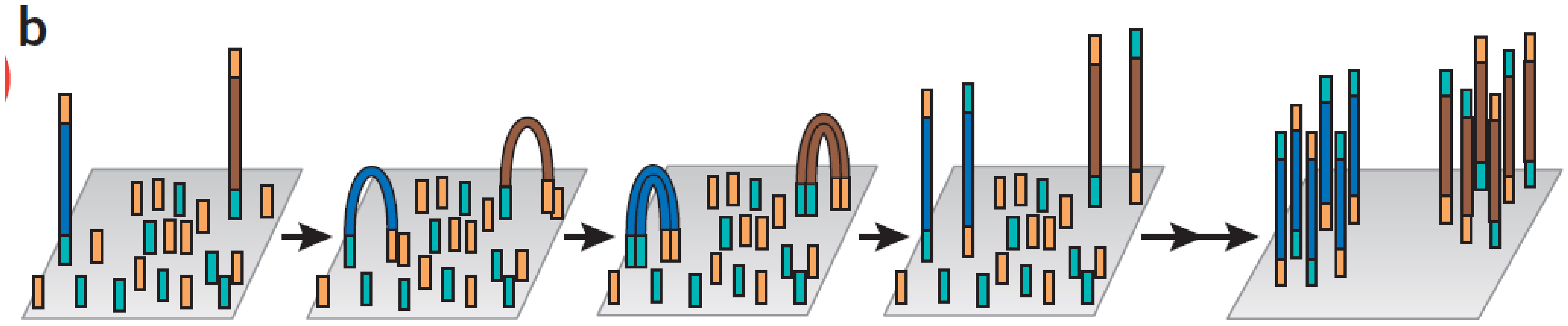
Denaturation at 94-96°C

Annealing at ~68°C

Elongation at ca. 72 °C

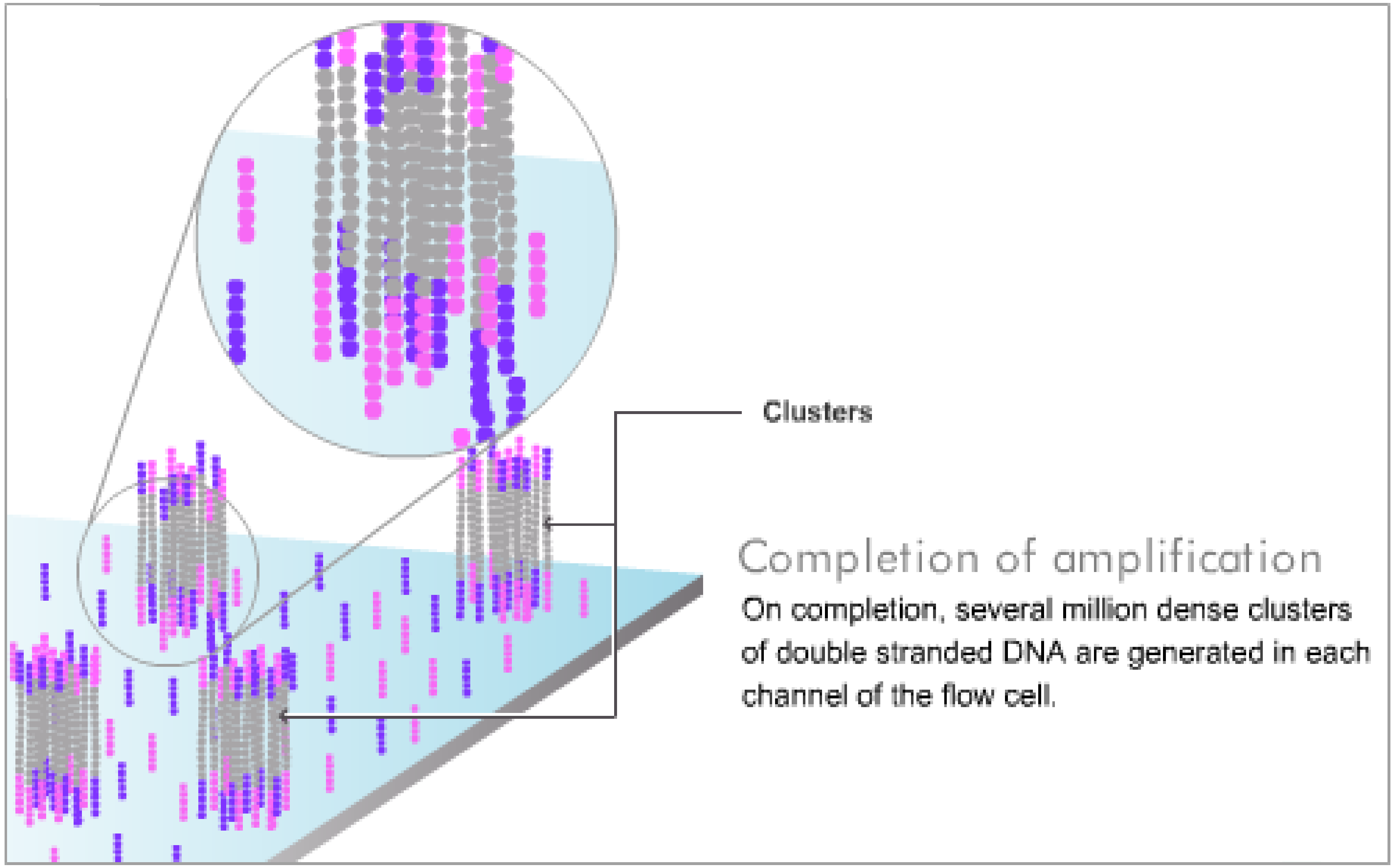
https://upload.wikimedia.org/wikipedia/commons/thumb/9/96/Polymerase_chain_reaction.svg/2000px-Polymerase_chain_reaction.svg.png

Illumina (Solexa) technology



Bridge amplification

Sequencing-By-Synthesis Demo

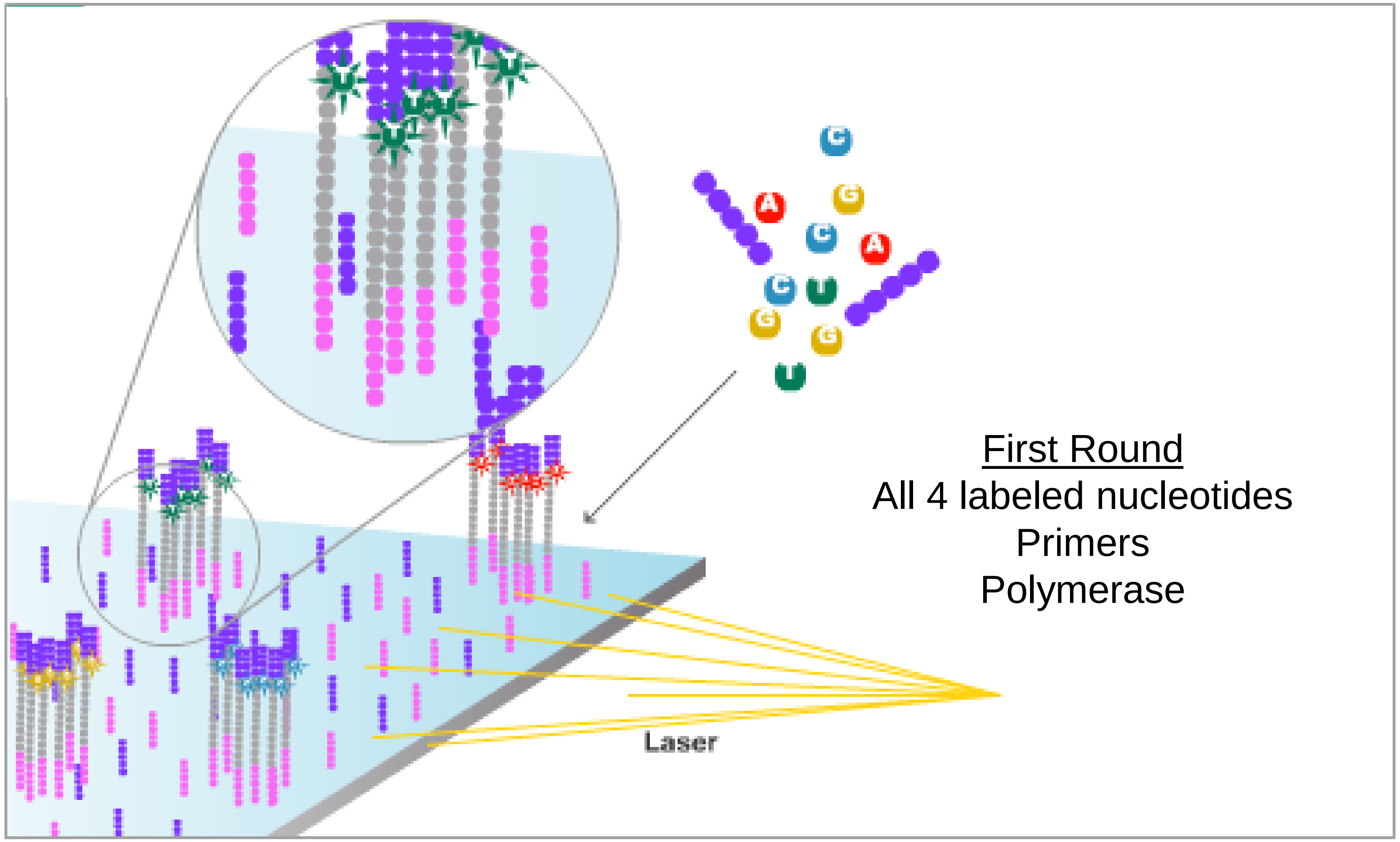


Completion of amplification

On completion, several million dense clusters of double stranded DNA are generated in each channel of the flow cell.

http://www.illumina.com/pages.ilmn?ID=203

Sequencing-By-Synthesis Demo



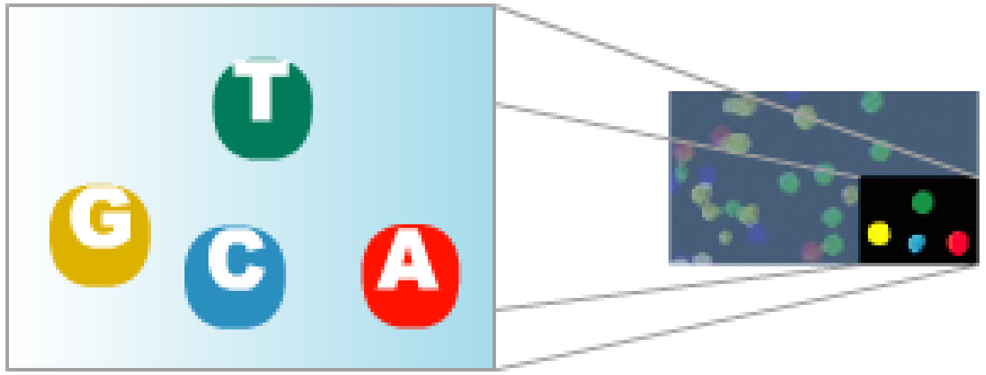
First Round

All 4 labeled nucleotides

Primers

Polymerase

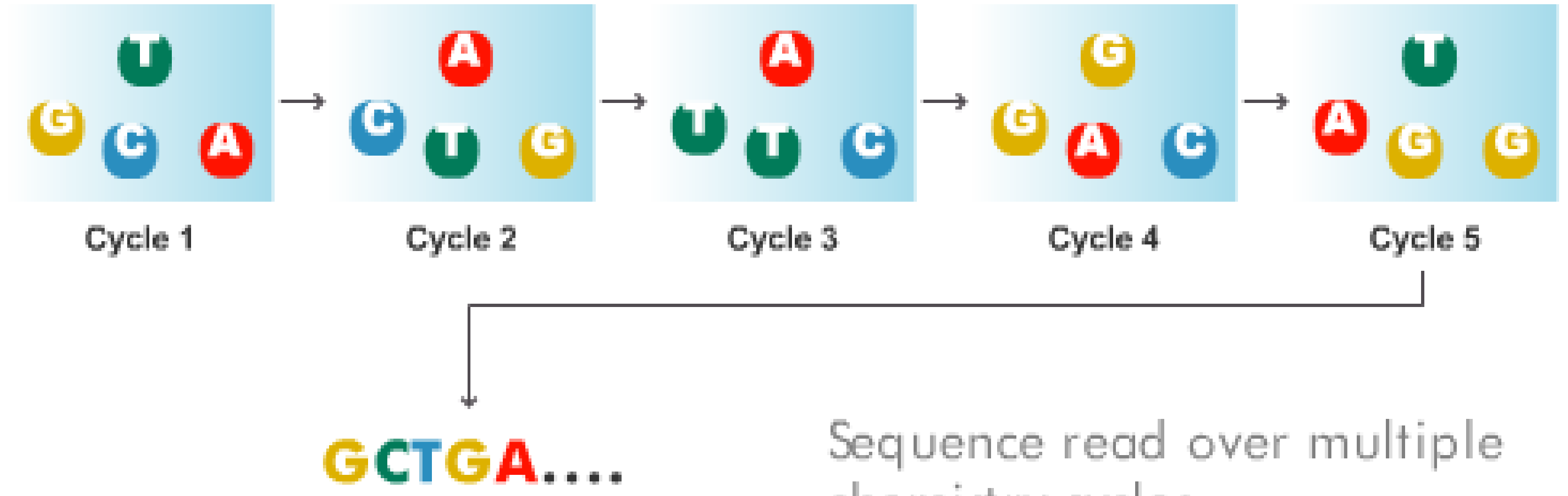
Sequencing-By-Synthesis Demo



1. Take image of first cycle

2. Remove fluorophore
3. Remove block on 3’ terminus

Sequencing-By-Synthesis Demo



Sequence read over multiple chemistry cycles

Repeat cycles of sequencing to determine the sequence of bases in a given fragment a single base at a time.

Phred Quality Score

$$q = -10\log_{10}(p)$$

- p=error probability for the base
- if p=0.01 (1% chance of error), then q=20
- p = 0.00001, (99.999% accuracy), q = 50
- Phred quality values are rounded to the nearest integer

Sequence qualities

- In most cases, the quality is poorest toward the ends, with a region of high quality in the middle

- Uses of sequence qualities
  - ‘Trimming’ of reads
    - Removal of low quality ends
  - Consensus calling in sequence assembly
  - Confidence metric for variant discovery

- Quality score can be taken into account while scoring the alignment; e.g. a mismatch with low quality is more likely seen than a mismatch with high quality

Mapping of reads

- Need to map the reads back to the original genome to detect variation compared to the reference or quantify expression level of a gene.
- We can always map the reads using pairwise alignment algorithm, however the quadratic-time complexity is infeasible for large number of reads and large reference genome.
- Need faster algorithms

Filter-based algorithms for mapping

- Note the difference between homology detection and mapping

- We assume that variations under the read-mapping setting are much rarer because
    - Mutation rate between individuals from the same species is low
    - Sequencing error rate is low

- We can thus assume for a given read (with relatively fixed length for a given sequencing platform), we only allow a given number of mismatches or indels.

Filter-based methods

- If we assume that we have a sequence with length *n*, and we allow up to *k* mismatches/indels.
- We can partition the sequence into *k*+d non-overlapping blocks, each block with length *n*/(*k*+d); we know that at least *d* block must match perfectly, because each mismatch/indel can disrupt at most one block.
- Scan the genome with each of these blocks; only initiate alignment when at least *d* block is found to be perfectly matched to the region

Gapped seeds

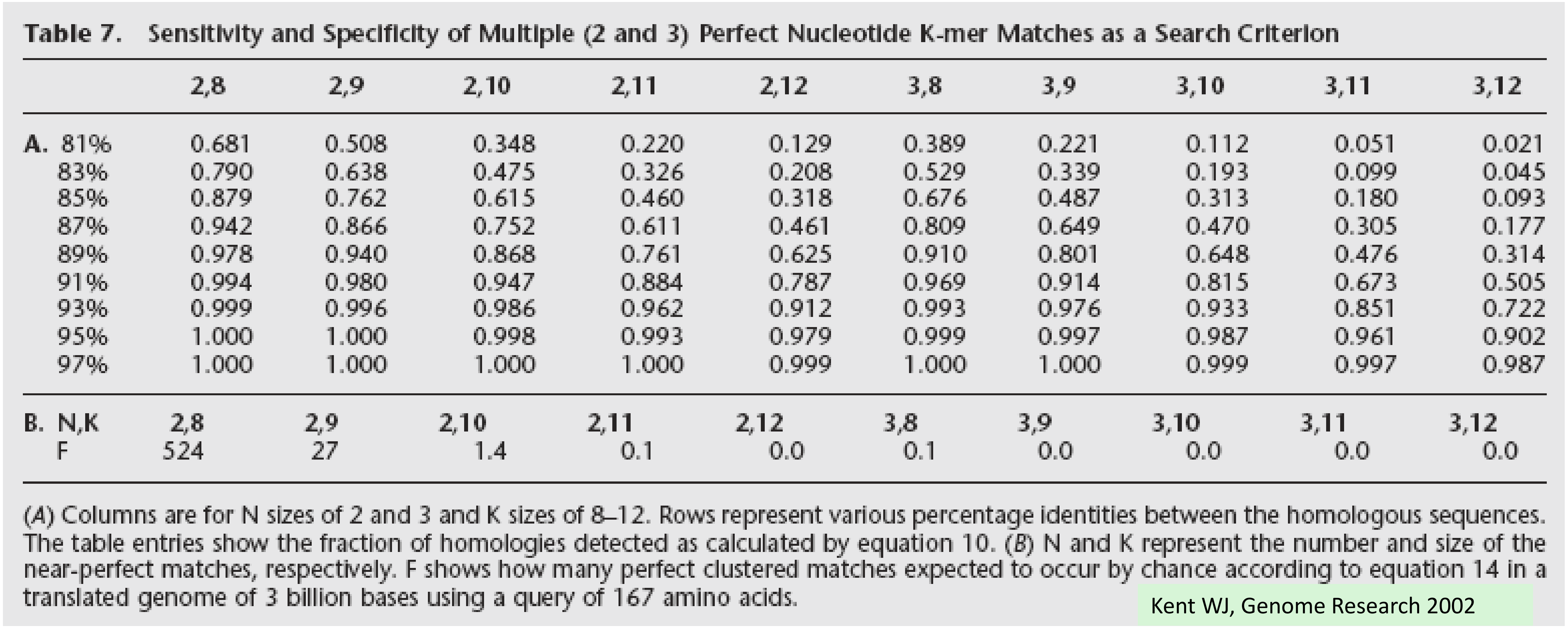


Table 7. Sensitivity and Specificity of Multiple (2 and 3) Perfect Nucleotide K-mer Matches as a Search Criterion

	2,8	2,9	2,10	2,11	2,12	3,8	3,9	3,10	3,11	3,12
A. 81%	0.681	0.508	0.348	0.220	0.129	0.389	0.221	0.112	0.051	0.021
83%	0.790	0.638	0.475	0.326	0.208	0.529	0.339	0.193	0.099	0.045
85%	0.879	0.762	0.615	0.460	0.318	0.676	0.487	0.313	0.180	0.093
87%	0.942	0.866	0.752	0.611	0.461	0.809	0.649	0.470	0.305	0.177
89%	0.978	0.940	0.868	0.761	0.625	0.910	0.801	0.648	0.476	0.314
91%	0.994	0.980	0.947	0.884	0.787	0.969	0.914	0.815	0.673	0.505
93%	0.999	0.996	0.986	0.962	0.912	0.993	0.976	0.933	0.851	0.722
95%	1.000	1.000	0.998	0.993	0.979	0.999	0.997	0.987	0.961	0.902
97%	1.000	1.000	1.000	1.000	0.999	1.000	1.000	0.999	0.997	0.987

B. N,K	2,8	2,9	2,10	2,11	2,12	3,8	3,9	3,10	3,11	3,12
F	524	27	1.4	0.1	0.0	0.1	0.0	0.0	0.0	0.0

(*A*) Columns are for N sizes of 2 and 3 and K sizes of 8–12. Rows represent various percentage identities between the homologous sequences. The table entries show the fraction of homologies detected as calculated by equation 10. (*B*) N and K represent the number and size of the near-perfect matches, respectively. F shows how many perfect clustered matches expected to occur by chance according to equation 14 in a translated genome of 3 billion bases using a query of 167 amino acids.

Kent WJ, Genome Research 2002

Filter-based methods

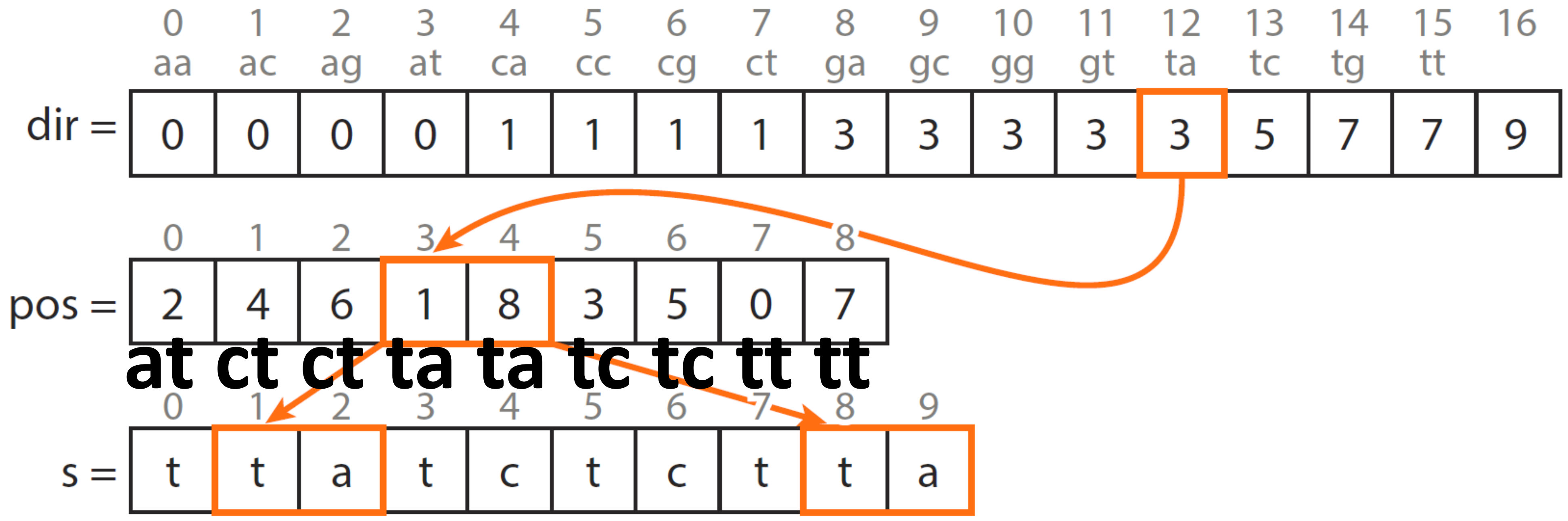
- An alternative method is to use fixed block size (*q*-gram or *q*-mer, or *q*-long word)
- We can partition an *n*-long sequence into *n*-*q*+1 overlapping *q*-grams
- Each mismatch/indel disrupt at most one *q*-gram; so we should have at least *n*-*q*+1-*kq* perfect *q*-gram matches to the reference genome
- Scan each of the *n*-*q*+1 *q*-grams against the reference, find regions with more than *n*-*q*+1-*kq* perfect hits as references

Fast-scanning of *q*-grams

- Fixed-length *q*-gram allows us to pre-construct arrays to facilitate fast identification of the *q*-grams

T = ttatctctta

All sorted 2-grams: at ct ct ta ta tc tc tt tt



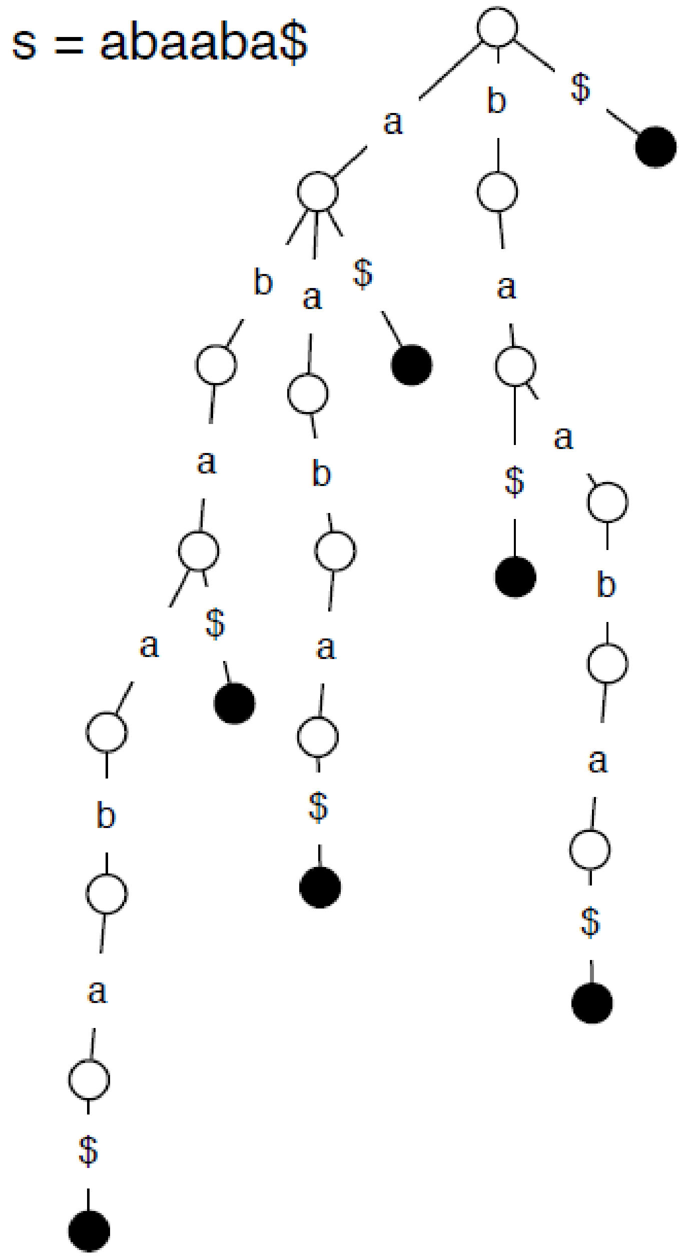
Index-based methods

- Suffix tree
- Suffix array
- Burrows-Wheeler Transformation

- All of them are conceptually equivalent
- Use large physical memory to store the precomputed index
- Sorted suffixes of the text
- Allows for ~O(|P|) search time, where |P| is the length of the query (independent of the text size)

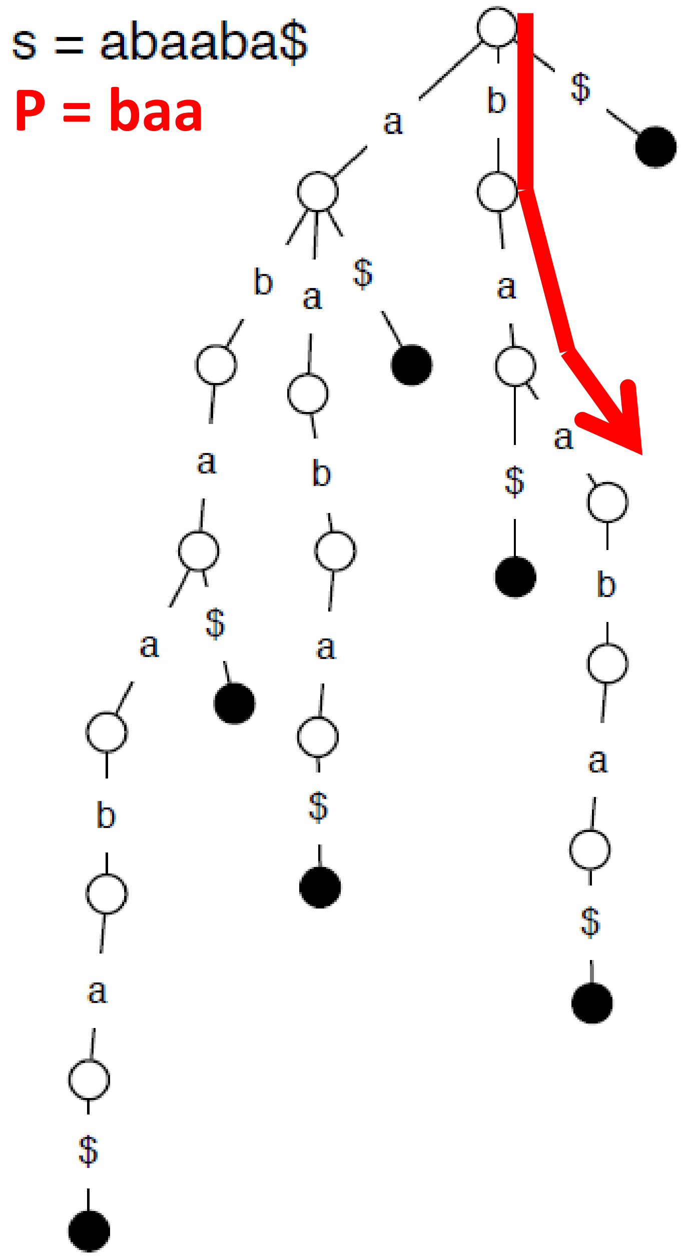
Suffix tree

- Edges of the suffix tree are labeled with letters from the alphabet Σ (say {A,C,G,T}).
- Every path from the root to a solid node represents a suffix of s.
- Every suffix of s is represented by some path from the root to a solid node.



Suffix tree

- Using suffix tree constructed on the text (usually the genome), we can determine whether a query string (usually a sequencing read) is a substring or suffix of the text
- By following the edges in the suffix tree
- We can also know the number of occurrences of the query string by counting the number of solid nodes in the subtree
- Time complexity is O(|P|), where |P| is the length of the query string (therefore independent of the text size)



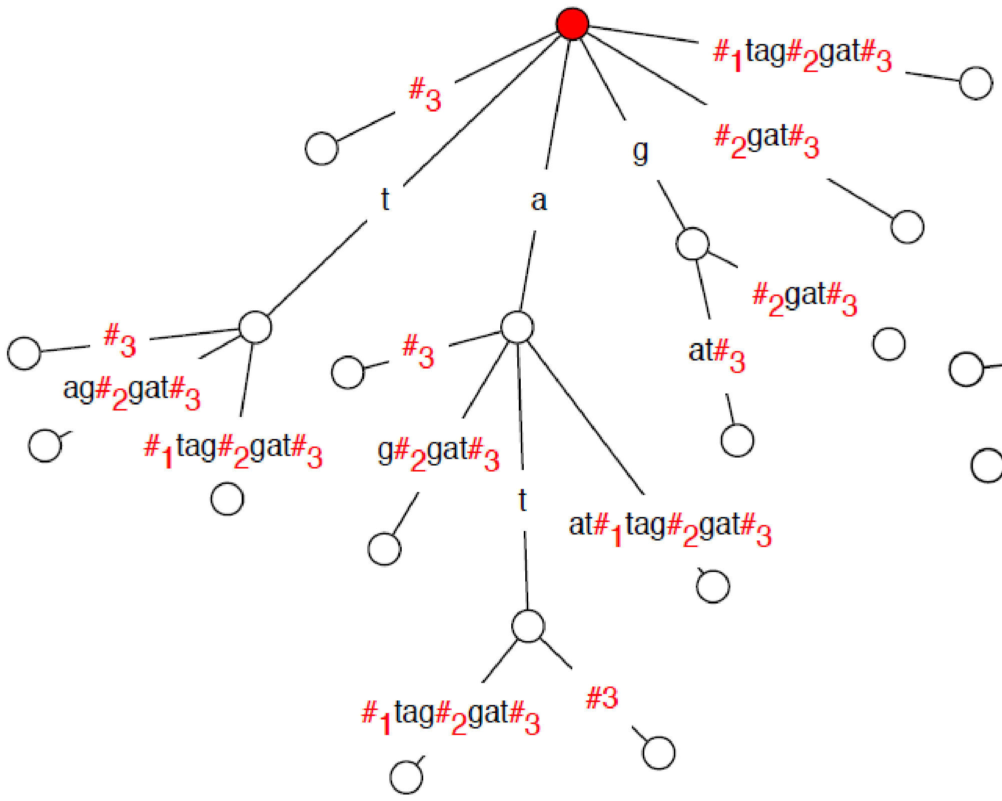
Generalized suffix tree

Goal. Represent a set of strings P = {s_1, s_2, s_3, ..., s_m}.

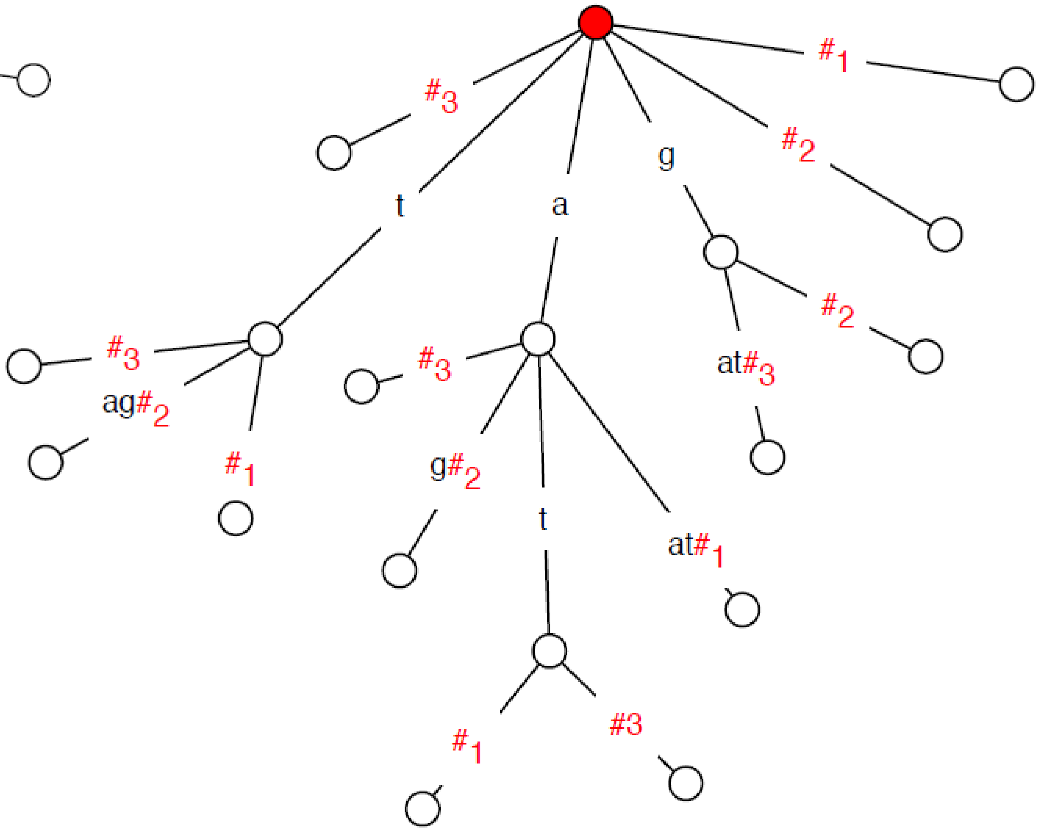
Example. att, tag, gat

Simple solution:

(1) build suffix tree for string aat#$_1$tag#$_2$gat#$_3$



(2) For every leaf node, remove any text after the first # symbol.



Generalized suffix tree

- Determine the strings in a database {*S1, S2, S3, ..., Sm*} that contain query string *P*:
  - Build generalized suffix tree for {*S1, S2, S3, ..., Sm*}
  - Follow the path for *q* in the suffix tree.
  - Suppose you end at node *u*: traverse the tree below *u*, and
  - output *i* if you find a string containing #*i*.

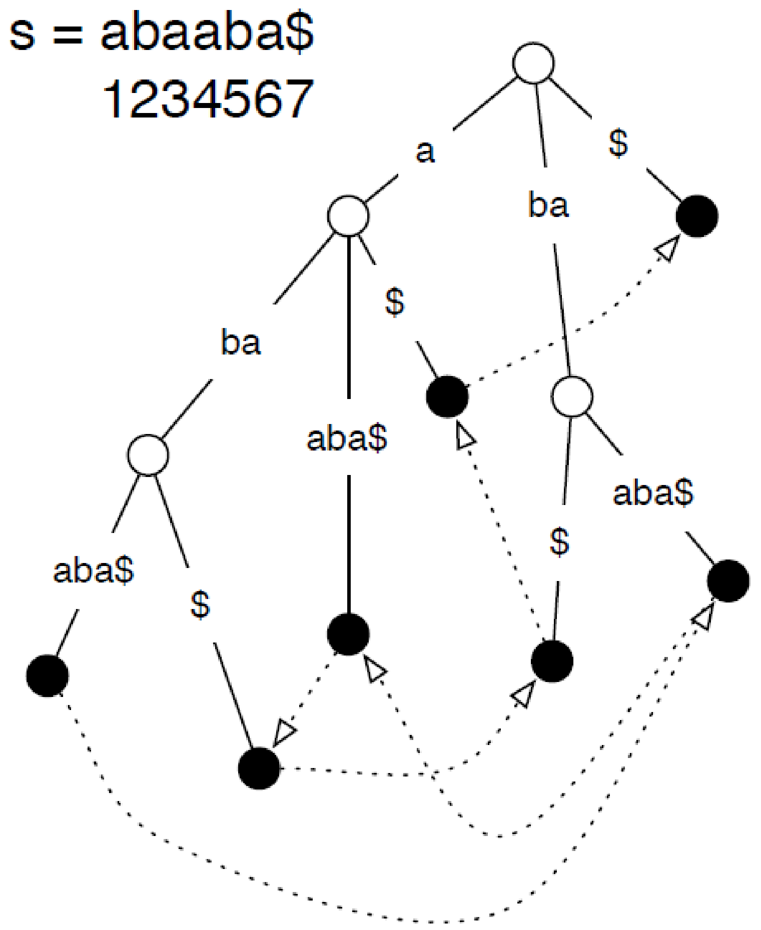
Space issue of suffix tree

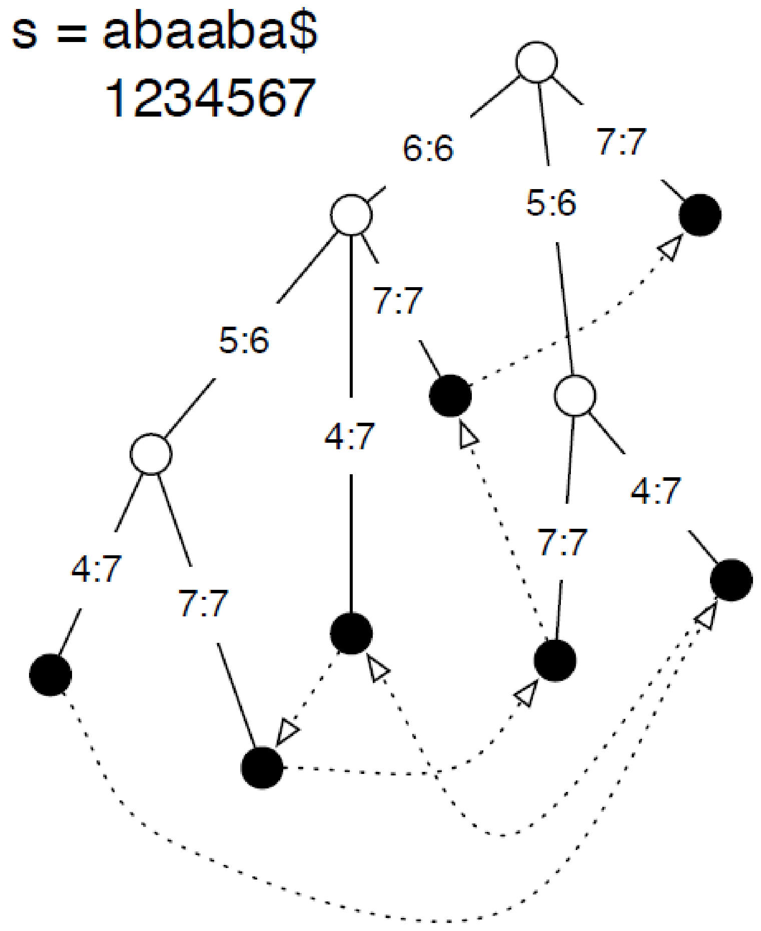
- Naïve representation of suffix tree would require O(n^2) space, where n is the size of the text. Under the read mapping setting, n is the size of the reference genome and is ~3G

- Because we need to represent every suffix of the text, so in the worst case the total number of nodes would be $n + (n - 1) + (n - 2) + … + 1$, which leads to O($n$^2) space complexity

- Needs a more compact representation for suffix tree

Space issue of suffix tree





- Compress paths where there are no choices.

- Represent sequence along the path using a range [i,j] that refers to the input string s.

- We have at most *n* solid nodes, and each internal node is at least a binary split

- Therefore the total number of nodes is O(*n*)

- Each node also requires O(1) space

Suffix array

- While both suffix trees and suffix arrays require O(*n*) space, suffix arrays are more space efficient. A recent suffix tree implementation requires **15-20 Bytes per character**. For suffix arrays, as few as **5 bytes** are sufficient (with some tricks).

- A moderate increase in search time from O(|P|) to O(|P| + log n). In practice this increase is **counterbalanced by better cache behavior**.

Suffix array

s = attcatg$

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

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

sort the suffixes alphabetically

$\longrightarrow$

the indices just "come along for the ride"

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

Suffix array

s = attcatg$

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

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

sort the suffixes alphabetically

$$\longrightarrow$$

the indices just “come along for the ride”

8
5
1
4
7
3
6
2

Suffix tree to suffix array

- Depth-first traversal of the suffix tree would return you the corresponding suffix array

- Linear time construction of suffix tree (Ukkonen's algorithm)

- Linear time construction of suffix array is also possible. See "*Linear Work Suffix Array Construction*" by Karkkainen *et al.*

Suffix array search

- Naïve approach: binary search with string comparison
- log(*n*) comparisons, each comparison would take O(|P|) time. So the complexity would be O(|P|*log(*n*))
- The time complexity is much higher than the one with suffix tree, which is O(|P|), we need to be smarter

Suffix array search

Consider further: binary search for suffixes with *P* as a prefix

Assume there's no **$** in *P*. So *P* can't be equal to a suffix.

Initialize $\underset{\substack{\uparrow \\ \text{“left”}}}{\boldsymbol{l}} = 0$, $\underset{\substack{\uparrow \\ \text{“center”}}}{\boldsymbol{c}} = \mathit{floor}(m/2)$ and $\underset{\substack{\uparrow \\ \text{“right”}}}{\boldsymbol{r}} = m$ (just past last elt of SA)

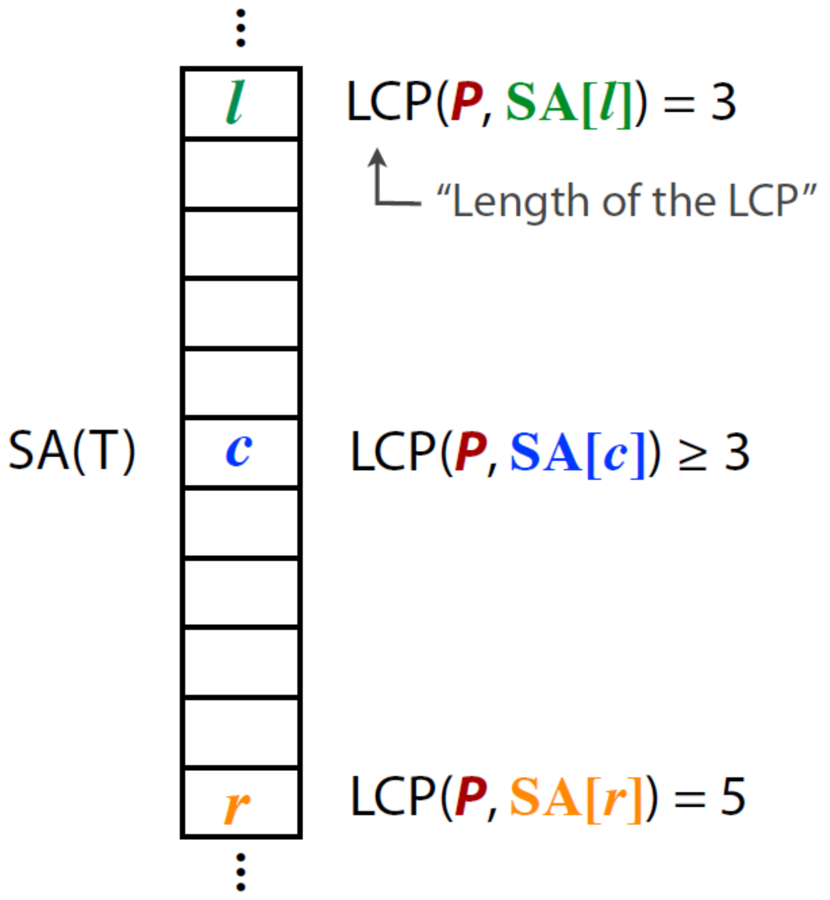
Notation: We'll use use **SA[*l*]** to refer to the suffix corresponding to suffix-array element ***l***. We could write *T*[**SA[*l*]**:], but that's too verbose.

Throughout the search, invariant is maintained:

$$\mathbf{SA}[\boldsymbol{l}] < P < \mathbf{SA}[\boldsymbol{r}]$$

Suffix array search using Longest Common Prefix (LCP)

Say we're comparing ***P*** to **SA[*c*]** and we've already compared *P* to **SA[*l*]** and **SA[*r*]** in previous iterations.



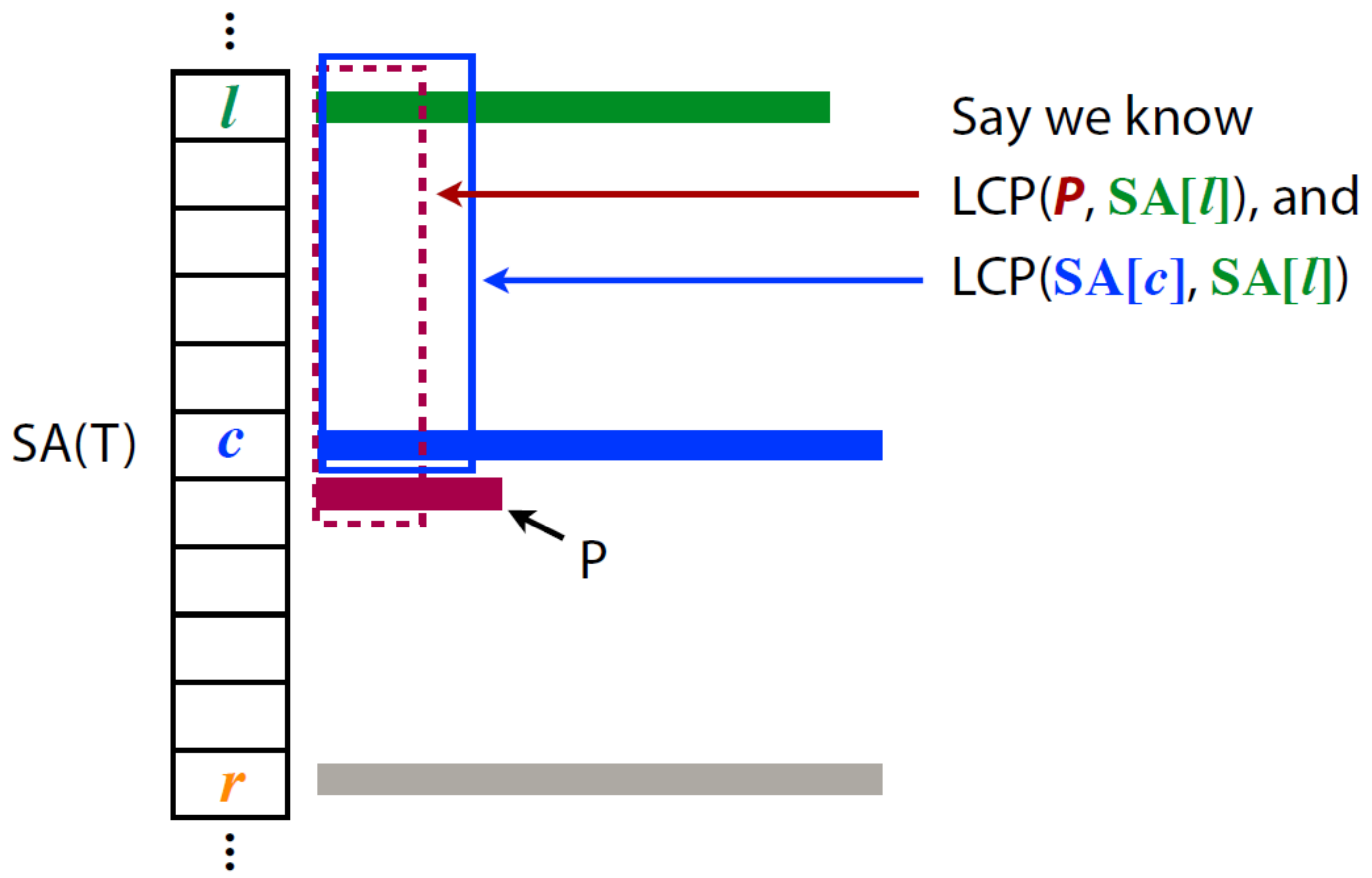
More generally:

LCP(***P***, **SA[*c*]**) ≥
min(LCP(***P***, **SA[*l*]**), LCP(***P***, **SA[*r*]**))

We can skip character comparisons

Suffix array search

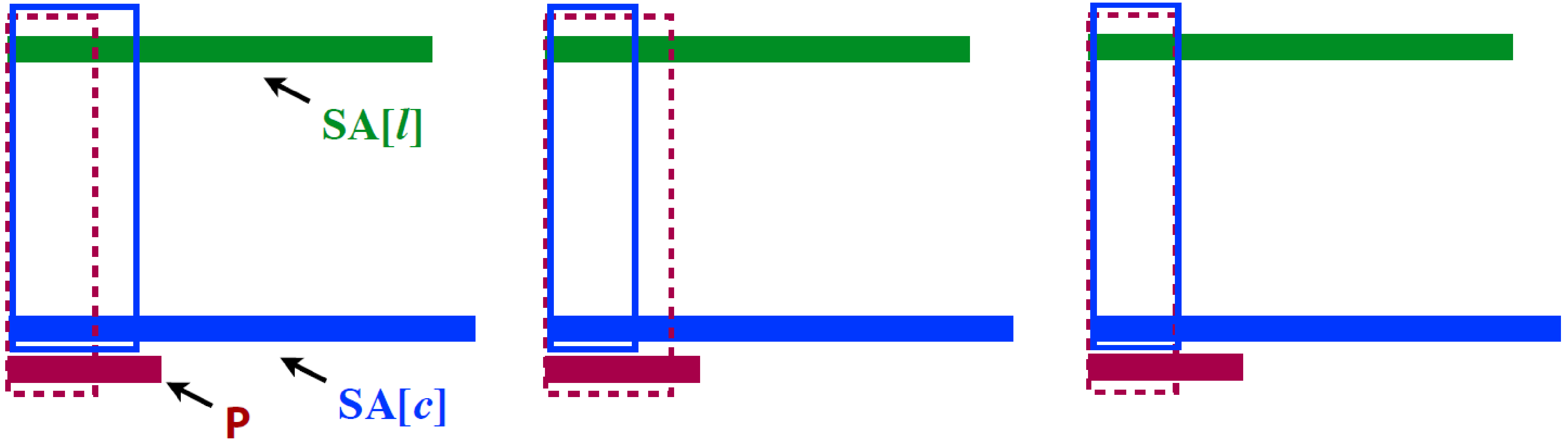
Take an iteration of binary search:



- When LCP(P, SA[*l*]) = LCP(P, SA[*r*]) we can always determine how to bisect by comparing P and SA[*c*] by skipping the first LCP(P, SA[*l*]) characters
- We are more interested in cases where LCP(P, SA[*l*]) != LCP(P, SA[*r*])
- Without loss of generality we assume that LCP(P, SA[*l*]) > LCP(P, SA[*r*])

Suffix array search

Three cases:



LCP(SA[c], SA[l]) $>$ LCP(P, SA[l])

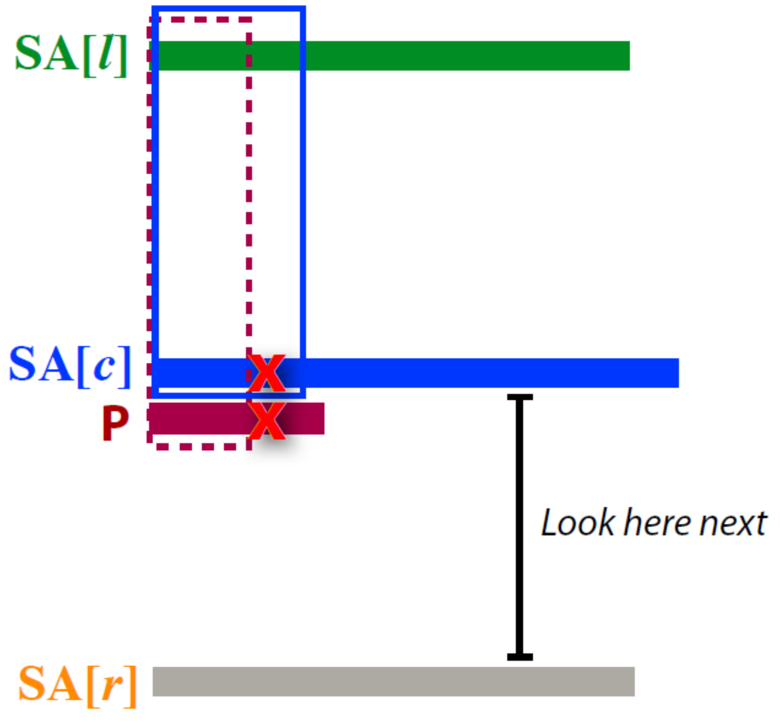
LCP(SA[c], SA[l]) $<$ LCP(P, SA[l])

LCP(SA[c], SA[l]) $=$ LCP(P, SA[l])

Suffix array search

No character comparison is required;
LCP(P, SA[*l*]) and LCP(P, SA[*r*]) remain unchanged;
LCP(P, SA[*l*]) and LCP(P, SA[*r*]) are non-decreasing

Case 1:



LCP(SA[*c*], SA[*l*]) > LCP(*P*, SA[*l*])

Next char of ***P*** after the LCP(***P***, SA[*l*]) must be *greater than* corresponding char of SA[*c*]

P > SA[*c*]

For the *i*th position (where LCP(P,SA[*l*]) < *i* <= LCP(SA[*l*], SA[c])):

- **P > SA[*l*]: assuming correctness of binary search**
- **SA[*l*] = SA[c]: LCP(SA[*l*], SA[c]) > LCP(P, SA[*l*])**

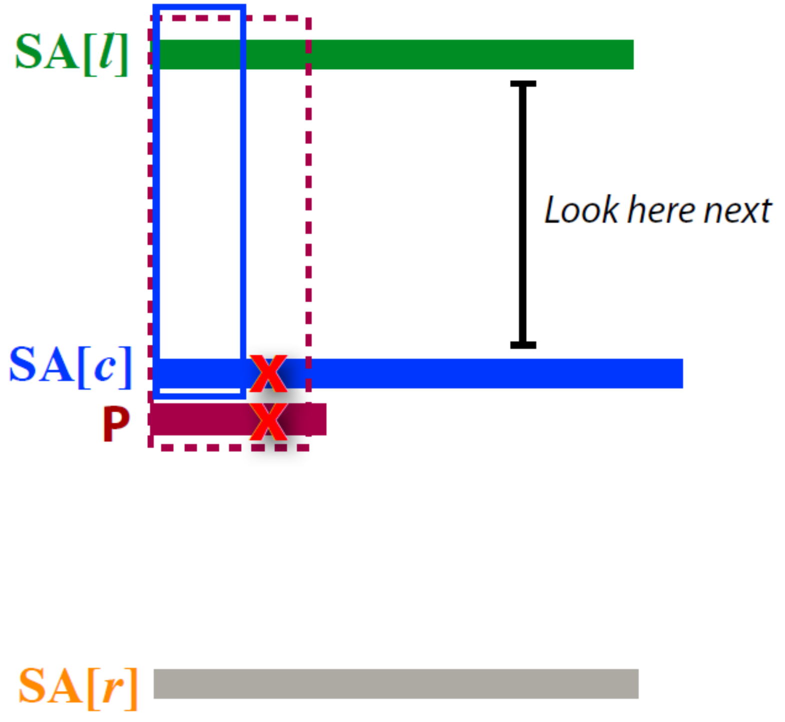
So:

- **P > SA[c]: we need to go to the lower half**

Suffix array search

No character comparison is required;
LCP(P,SA[*l*]) remains the same
LCP(P,SA[*r*]) is set to LCP(SA[*l*], SA[*c*]);
LCP(SA[*l*], SA[*c*]) >= LCP(P,SA[*r*]);
LCP(P, SA[*l*]) and LCP(P, SA[*r*]) are non-decreasing

Case 2:



LCP(SA[*c*], SA[*l*]) <
LCP(*P*, SA[*l*])

Next char of SA[*c*] after LCP(SA[*c*], SA[*l*])
must be *greater than* corresponding char of ***P***

P < SA[*c*]

For the *i*th position (where LCP(P,SA[*l*]) < *i* <= LCP(SA[*l*], SA[c])):

- **P = SA[*l*]: LCP(SA[*l*], SA[c]) > LCP(P, SA[*l*])**
- **SA[*l*] < SA[c]: lexicographical sorting of suffixes**

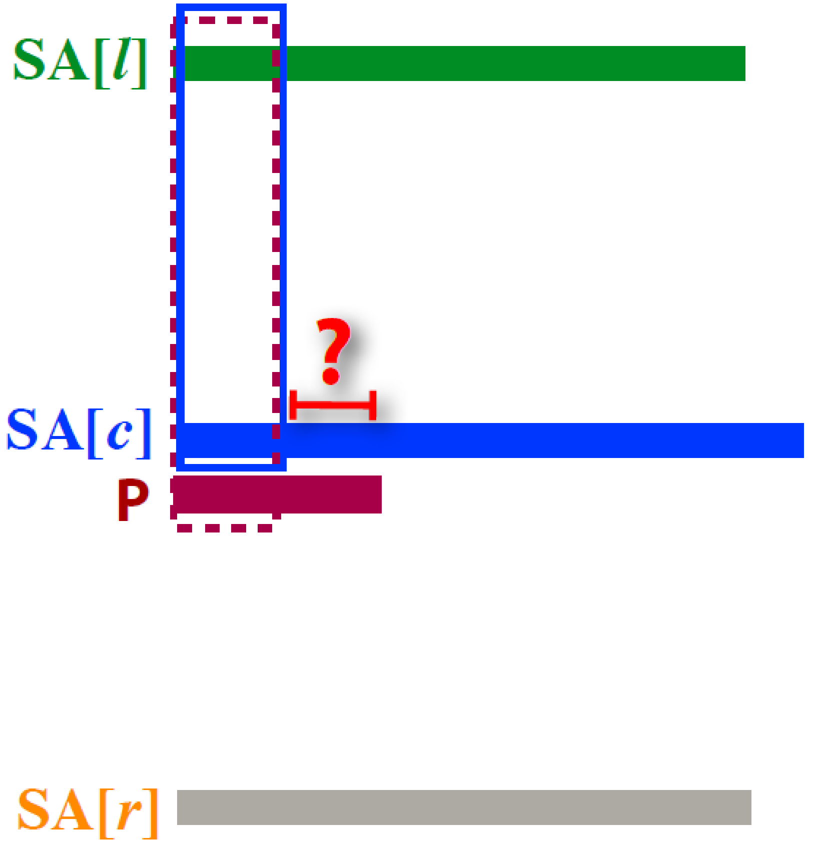
So:

- **P < SA[c]: we need to go to the upper half**

Suffix array search

Needs character comparison;
Either LCP[P, SA[*l*]] or LCP[P,SA[*r*]] is increased or remains the same; the other one remains the same;
LCP(P, SA[*l*]) and LCP(P, SA[*r*]) are non-decreasing

Case 3:



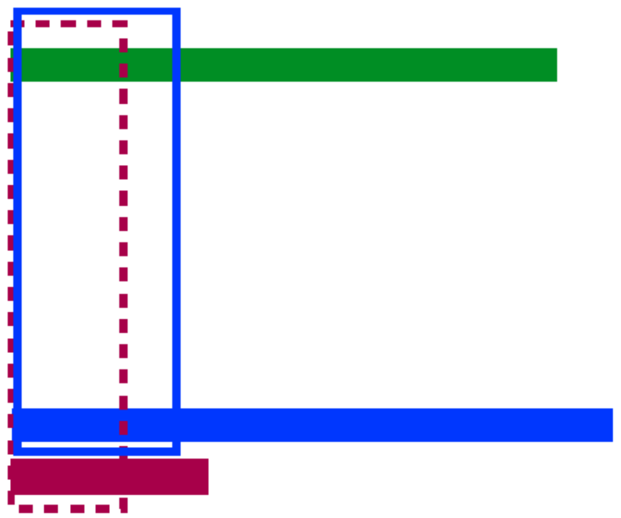
Must do further character comparisons between **P** and **SA[*c*]**

Each such comparison either:

(a) mismatches, leading to a bisection

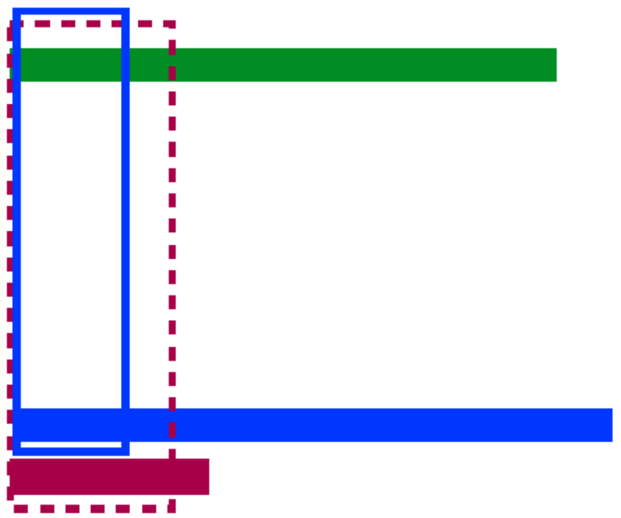
(b) matches, in which case LCP(***P***, **SA[c]**) grows

Suffix array search



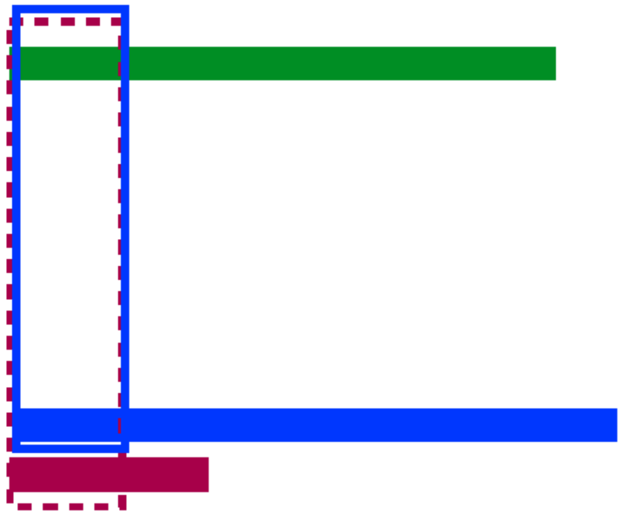
$\text{LCP}(SA[c], SA[l]) > \text{LCP}(P, SA[l])$

Bisect right!



$\text{LCP}(SA[c], SA[l]) < \text{LCP}(P, SA[l])$

Bisect left!

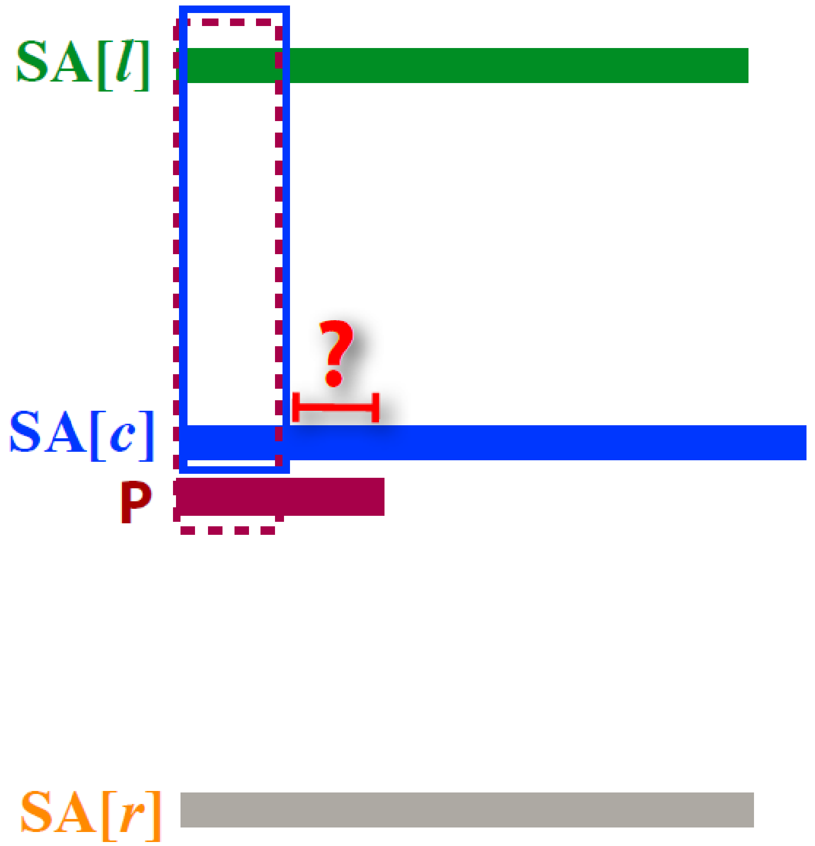


$\text{LCP}(SA[c], SA[l]) = \text{LCP}(P, SA[l])$

Compare some characters, then bisect!

Comparisons performed

Case 3:

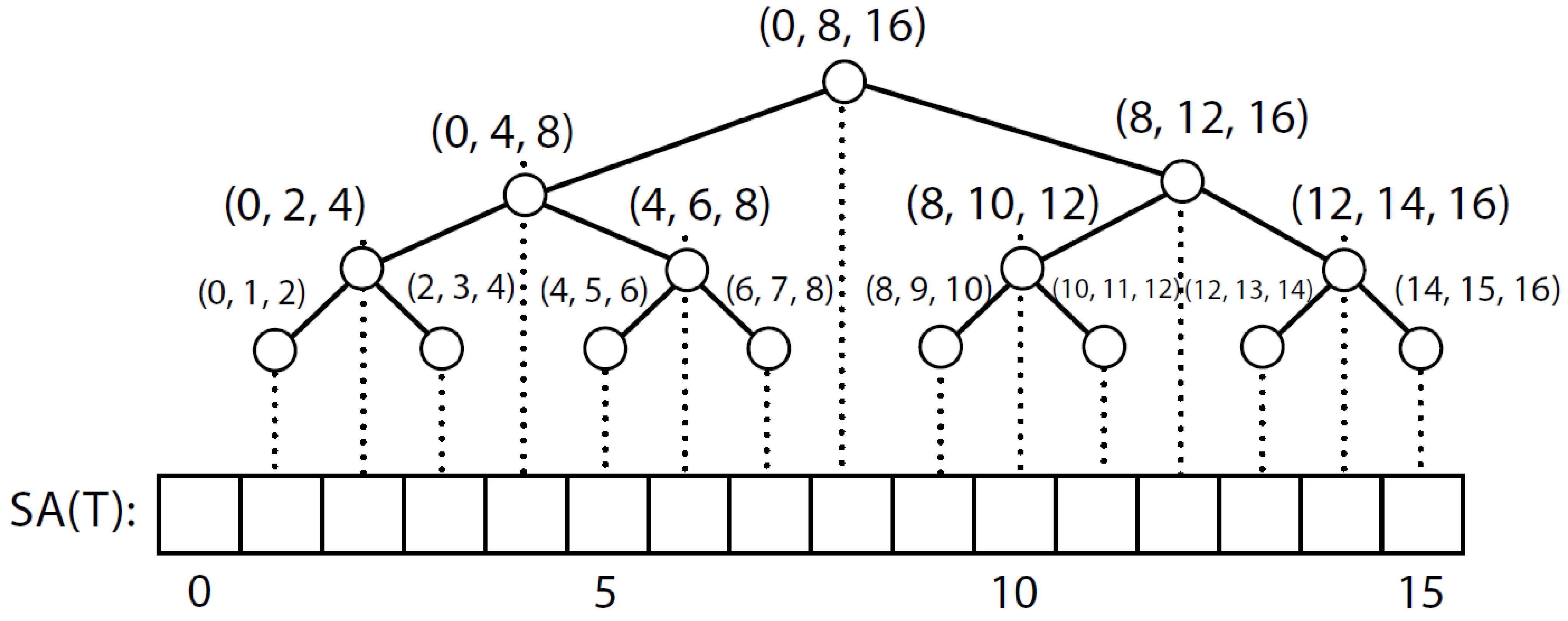


- Let max(LCP(P, SA[*l*]), LCP(P, SA[*r*])) be *M*
- If we have LCP(P, SA[*l*]) = LCP(P, SA[*r*]), then we do direct comparison to decide where to bisect, *k* comparisons will increase *M* by *k* – 1
- Note that without loss of generality we assume LCP(P, SA[*l*]) > LCP(P, SA[*r*]), so M= LCP(P, SA[*l*])
- In the first case, when we decide to bisect left, LCP(P, SA[*c*]) is at least *M*, which means that the new LCP(P, SA[*r*]) is also at least *M*, and *k* comparisons will increase *M* by *k* – 1
- In the second case, when we decide to bisect right, LCP(P, SA[*c*]) us going to be used as the new LCP(P, SA[*l*]) and is also at least *M*; and *k* comparisons will increase *M* by *k* - 1
- In summary, M is non-decreasing as neither LCP(P, SA[*l*]) nor LCP(P, SA[*r*]) is deceasing; *k* comparisons will increase *M* by *k* – 1, and the search terminates when *M* reaches |P|

Suffix array search complexity

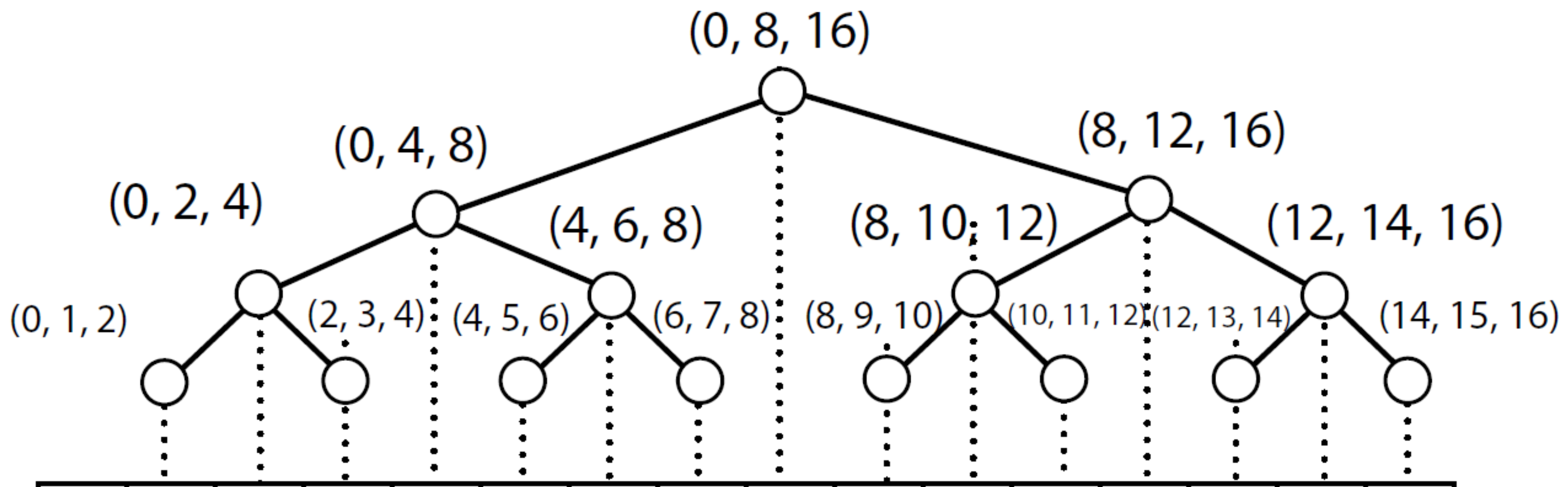
- The total number of character comparison is |P| + *x*, where *x* is the number of times that comparison is taken (either LCP(P, SA[l]) = LCP(P, SA[r]) or the 3rd case we discussed before)
- For the 1st and 2nd cases only constant number of operation is required, so they are also bounded by *x*
- Since it is a binary search, the number of steps taken, *x*, is bounded by log(n). Thus the time complexity is O(|P| + log(n))

Storing the LCP information



- **O(n) leaf nodes, so O(n) nodes.**
- **We can pre-calculate and store the LCP information efficiently.**

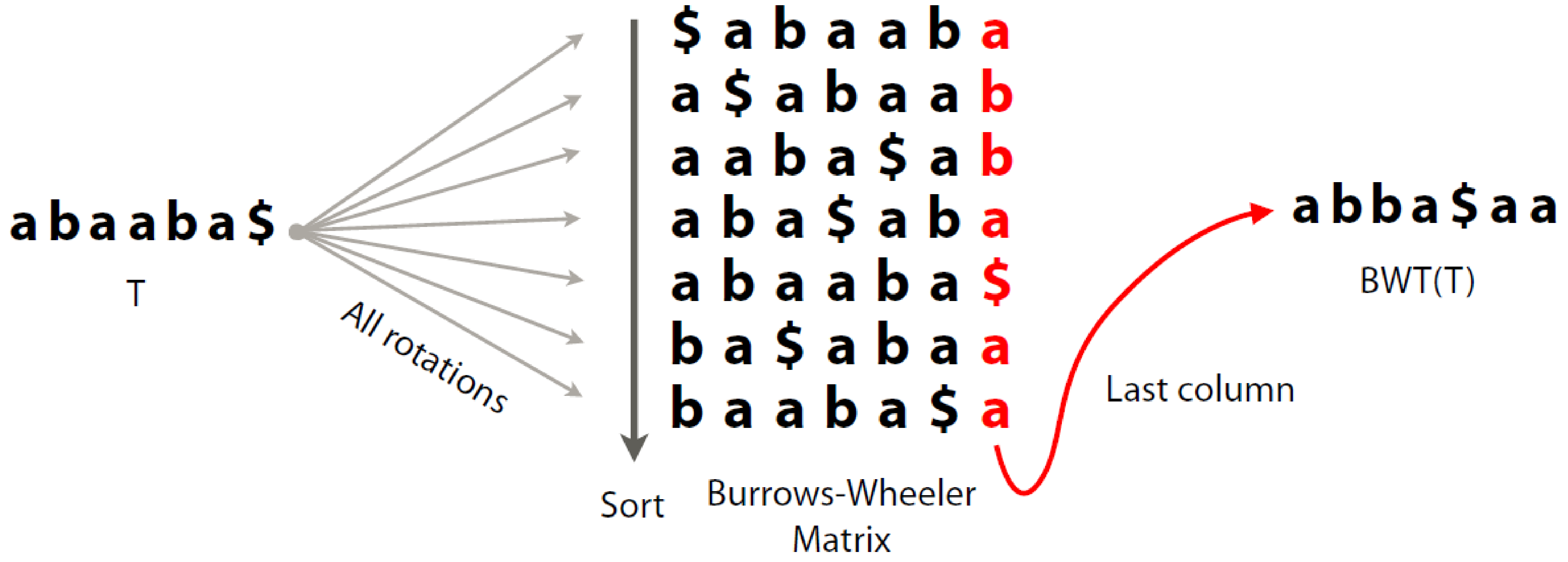
Storing the LCP information



SA(T):	15	14	7	0	10	3	12	5	8	1	11	4	13	6	9	2
LCP1(T):	0	1	8	1	5	1	3	0	7	0	4	0	2	0	6	
LCP_LC(T):	0	0	8	0	5	1	3	0	7	0	4	0	2	0	6	
LCP_CR(T):	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	
	0					5					10					15

Burrows-Wheeler Transformation

Reversible permutation of the characters of a string, used originally for compression



BWT and suffix array

BWM bears a resemblance to the suffix array

```
$ a b a a b a
a $ a b a a b
a a b a $ a b
a b a $ a b a
a b a a b a $
b a $ a b a a
b a a b a $ a
```

BWM(T)

6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

SA(T)

Sort order is the same whether rows are rotations or suffixes

BWT LF mapping property

Give each character in *T* a rank, equal to # times the character occurred previously in *T*. Call this the *T-ranking*.

$$a_0 \; b_0 \; a_1 \; a_2 \; b_1 \; a_3 \; \$$$

Now let's re-write the BWM including ranks...

BWT LF mapping property

BWM with T-ranking:

$$\begin{array}{ccccccc}
F & & & & & & L \\
\$ & a_0 & b_0 & a_1 & a_2 & b_1 & a_3 \\
a_3 & \$ & a_0 & b_0 & a_1 & a_2 & b_1 \\
a_1 & a_2 & b_1 & a_3 & \$ & a_0 & b_0 \\
a_2 & b_1 & a_3 & \$ & a_0 & b_0 & a_1 \\
a_0 & b_0 & a_1 & a_2 & b_1 & a_3 & \$ \\
b_1 & a_3 & \$ & a_0 & b_0 & a_1 & a_2 \\
b_0 & a_1 & a_2 & b_1 & a_3 & \$ & a_0
\end{array}$$

Look at first and last columns, called *F* and *L*

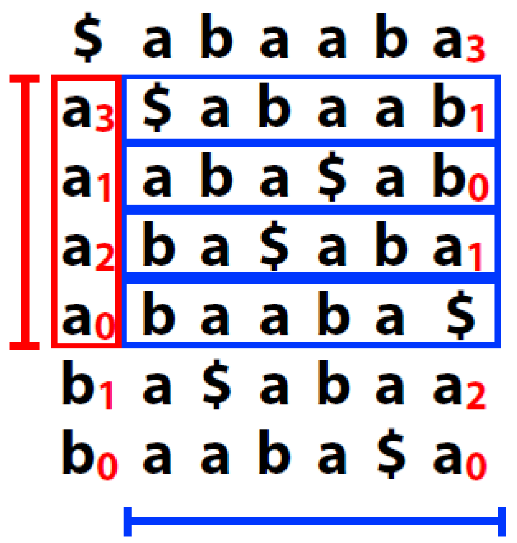
And look at just the **a**s

as occur in the same order in *F* and *L*. As we look down columns, in both cases we see: a_3, a_1, a_2, a_0

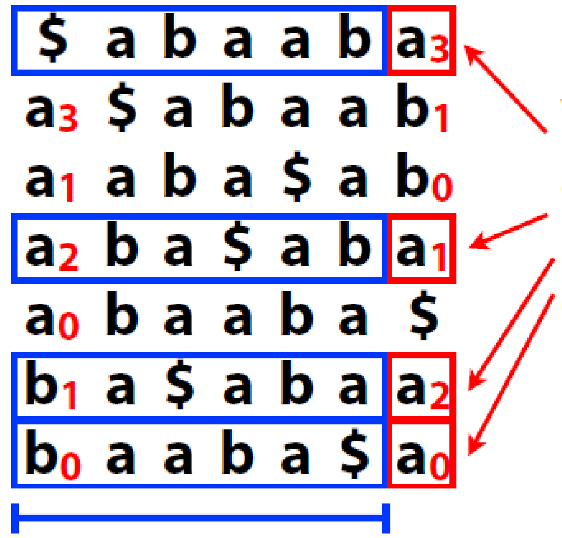
BWT LF mapping property

Why does the LF Mapping hold?

Why are these **a**s in this order relative to each other?



They're sorted by right-context



They're sorted by right-context

Why are these **a**s in this order relative to each other?

Occurrences of *c* in *F* are sorted by right-context. Same for *L*!

Whatever ranking we give to characters in *T*, rank orders in *F* and *L* will match

BWT search

We need the ranking of the letter in the last column to determine the interval in the first column.

	F	*L*	
	$	$\mathbf{a}_0$	
	$\mathbf{a}_0$	$\mathbf{b}_0$	
	$\mathbf{a}_1$	$\mathbf{b}_1$	$\longleftarrow$ Which BWM row *begins* with $\mathbf{b}_1$?
	$\mathbf{a}_2$	$\mathbf{a}_1$	
	$\mathbf{a}_3$	**$**	
	$\mathbf{b}_0$	$\mathbf{a}_2$	
row 6 $\rightarrow$	$\mathbf{b}_1$	$\mathbf{a}_3$	

Skip row starting with **$** (1 row)

Skip rows starting with **a** (4 rows)

Skip row starting with $\mathbf{b}_0$ (1 row)

Answer: row 6

FM index

- In the previous example, we need to know the order of *b* in order to calculate its positions at the first column.
- We can always walk through the entire list to figure out, but it would be too slow.
- Use FM index: build a $|\Sigma|*n$ matrix, for each letter at position *i*, it stores how many such letter has occurred in the BWT.
- We can reduce the space of the FM index by an arbitrary factor *k*, but it would increase the search time by a factor of *k*

BWT search

BWTSearch(aba) Start from the **end** of the pattern

Step 1: Find the range of "a"s in the first column

Step 2: Look at the same range in the last column.

Step 3: "b" is the next pattern character. Set B = the LF mapping entry for b in the first row of the range.
Set E = the LF mapping entry for b in the last + 1 row of the range.

Step 4: Find the range for "b" in the first row, and use B and E to find the right subrange within the "b" range.

LF Mapping

Σ

BWT(unabashable)	$	a	b	e	h	l	n	s	u
$unabashable	0	0	0	0	0	0	0	0	0
abashable$un	0	0	$\boxed{0}$	1	0	0	0	0	0
able$unabash	0	0	0	1	0	0	1	0	0
ashable$unab	0	0	0	1	1	0	1	0	0
bashable$una	0	0	$\boxed{1}$	1	1	0	1	0	0
ble$unabasha	0	1	1	1	1	0	1	0	0
e$unabashabl	0	2	1	1	1	0	1	0	0
hable$unabas	0	2	1	1	1	1	1	0	0
le$unabashab	0	2	1	1	1	1	1	1	0
nabashable$u	0	2	2	1	1	1	1	1	0
shable$unaba	0	2	2	1	1	1	1	1	1
unabashable$	0	3	2	1	1	1	1	1	1
	1	3	2	1	1	1	1	1	1

BWT Search

- It would take O(*kl*) time

- *k* in practice is a constant depending on available memory resource, and usually take on a value around 128

Suffix tree, suffix array, and BWT

- Suffix Array = suffix numbers obtained by traversing the leaf nodes of the (ordered) Suffix Tree from left to right

- BWT can be constructed from Suffix Array by taking the previous character of the sorted suffixes

- Space efficiency: BWT > Suffix array > Suffix tree
- Search efficiency: Suffix tree > Suffix array > BWT

- Recall that modern memory architecture allows prefetching, so using less memory will in practice reduce the running time

- BWT remains the choice for most current mapping implementations

Accounting for mismatch and indels

- Set an upper limit for the number of mismatches and indels allowed for the alignment.
- Use backtracking when the search reaches a dead end.
- The search time would be exponential in terms of the number of mismatches or indels allowed.
- It is usually doable given that current technologies have very low error rate (~0.1-1%)