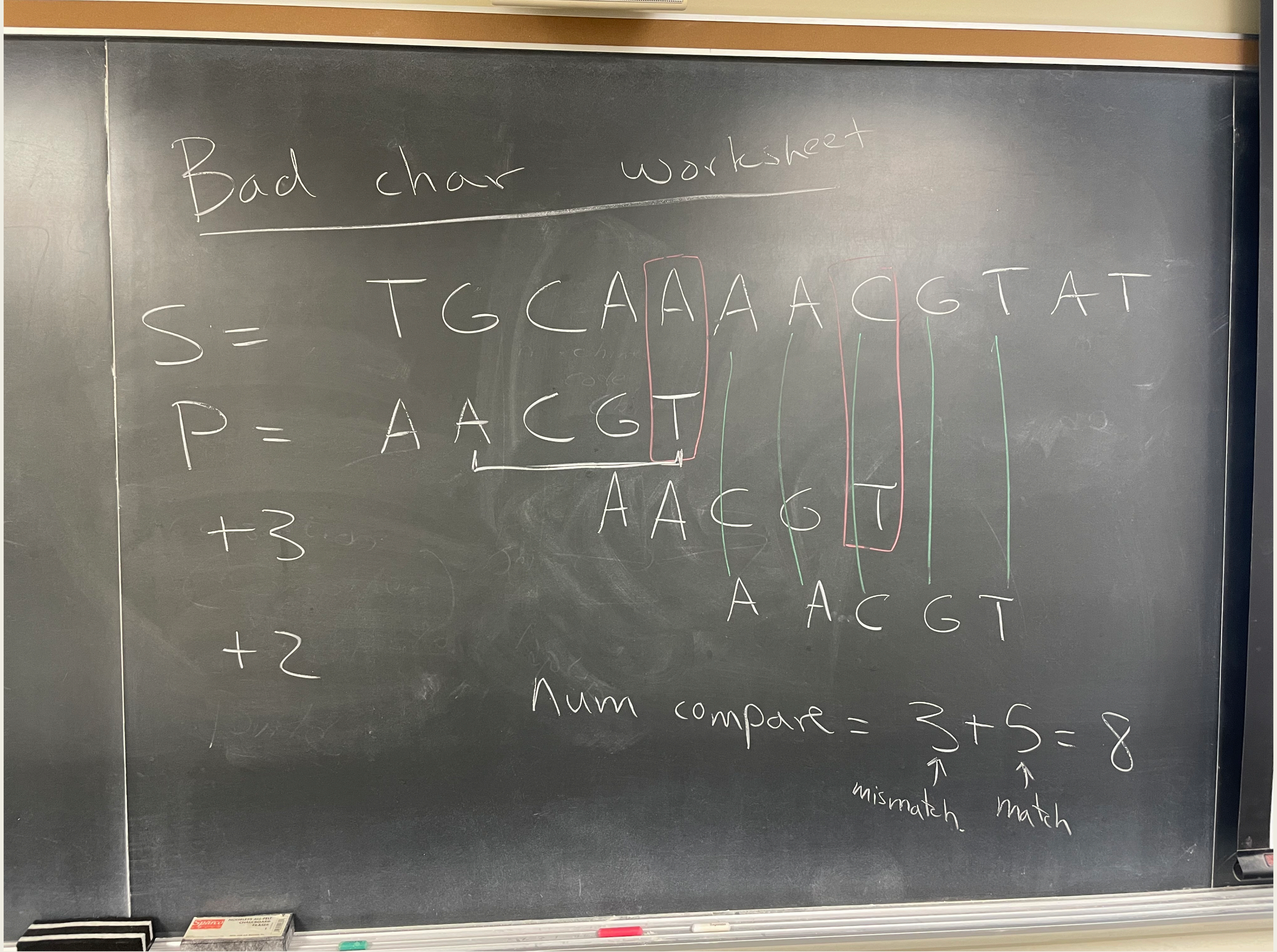
CS 364 COMPUTATIONAL BIOLOGY

Sara Mathieson
Haverford College

Outline

- String search: finish Boyer-Moore algorithm
- Sequencing pipeline overview
- Read mapping
- Burrows-Wheeler Transform (BWT)

Boyer-Moore Algorithm



Bad char worksheet

```
S =    T G C A A A A C G T A T
P =    A A C G T
+3           A A C G T
+2               A A C G T
```

num compare = $\underset{\uparrow \text{mismatch}}{3} + \underset{\uparrow \text{match}}{5} = 8$

```
S = AAATAAATAAATAAAT
    AAAA
        AAAA
```

skip

Good Case

```
S = AAAAAAAAAAAATAAA
    TAAA
     TAAA
```

No skip

```
S = AAAAAAAAAAAATAAA
    AAAT
     AAAT
```

No skip

Bad Cases

Worst-case time complexity proportional to *nm*, where *n* is size of S and *m* is size of the pattern, P. However, in practice *average* time complexity is very good.

Can we do better in these worst case situations?

Idea: there is information in “matches so far”

```
S = AAAAAAAAAAAATAAA
    TAAA
```

Found a mismatch at the left most position, where the pattern has a T but we know that S has AAA to the right of that, so we could never match in that region and can skip over it.

```
S = AAAAAAAAAAAAAAAA
 +4 ───▶    TAAA
```

Good Suffix Rule: Consider all the possible points in P where mismatch can happen. For each kind of mismatch, look left and see if the mismatch pattern up to the current point can be found in P again.

```
  123456789
S:TATTCGGTT
P:GCGACG
```

^ACG

```
  123456789
S:TATTCGGTT
P:   GCGACG
```

Notation: We use ^X to mean “not X”

So ^A = C, G or T

^ACG = CCG, GCG or TCG etc…

```
  123456789
S:TATTCGGTT
P:GCGACG
```

(Bold in original: the T at position 4 of S and the A at position 4 of P.)

```
  123456789
S:##^ACG###
P:GCGACG
```

```
   123456789
S:##^ACG###
P:  GCGACG        +1
```

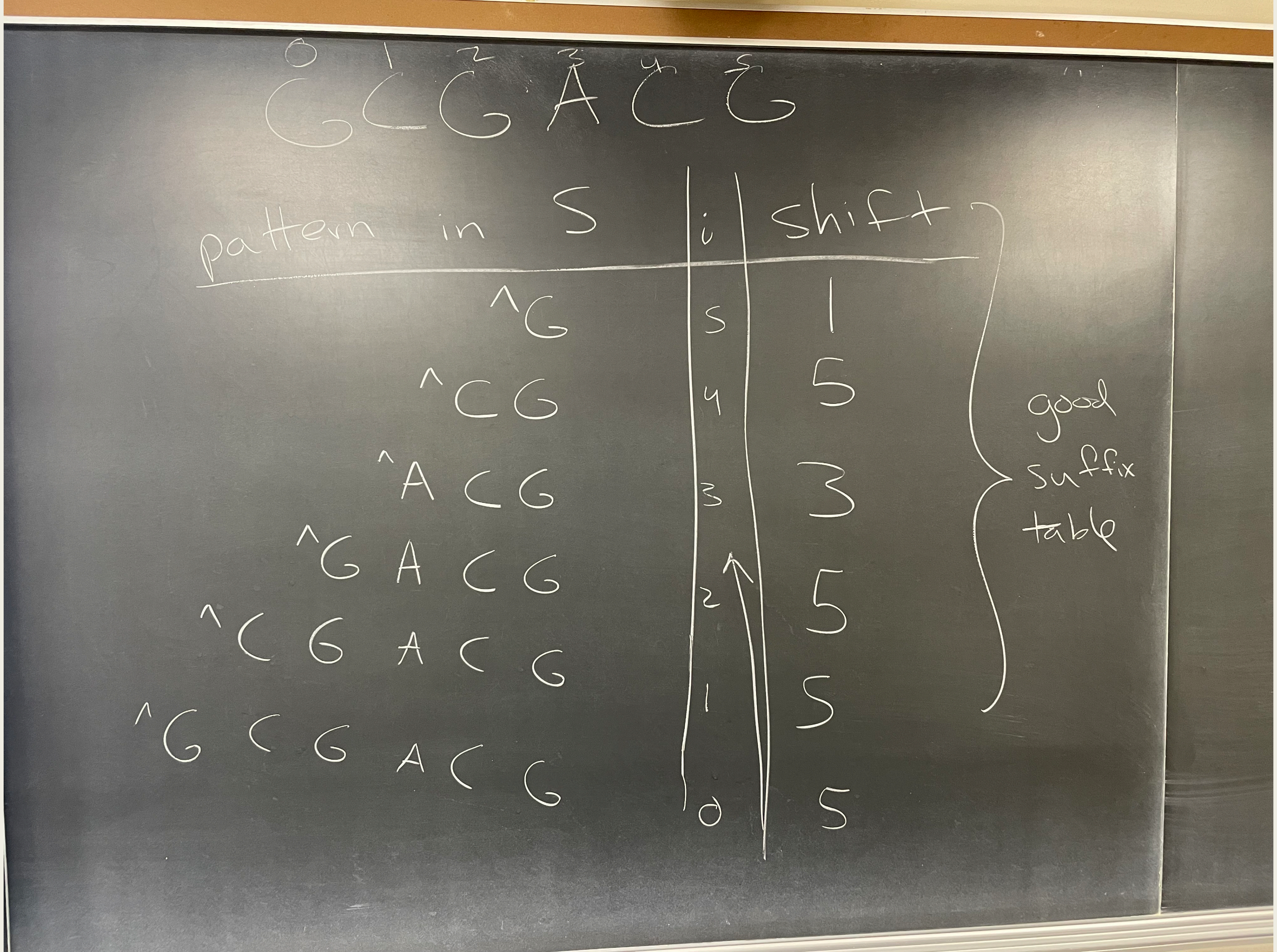
```
  123456789
S:##^ACG###
P:  GCGACG       +2
```

```
   123456789
S:##^ACG###
P:    GCGACG       +3
```

```
  123456789
S:##^ACG###
P:    GCGACG      +3
```

All possible mismatch patterns for P = “GCGACG”

Position of first mismatch of P, scanning from right to left	Implied pattern in S	Action
6	`^G`	move 1
5	`^CG`	move 5
4	`^ACG`	move 3
3	`^GACG`	move 5
2	`^CGACG`	move 5
1	`^GCGACG`	move 5



$$\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \\ G & C & G & A & C & G \end{array}$$

pattern in S	i	shift
^G	5	1
^C G	4	5
^A C G	3	3
^G A C G	2	5
^C G A C G	1	5
^G C G A C G	0	5

good suffix table

Exercise: P = “TAAAA”

Position of first mismatch of P, scanning from right to left	Implied pattern in S	Action to be taken
P[5]	`^A`	increase offset by 4
P[4]	`^AA`	increase offset by 3
P[3]	`^AAA`	increase offset by 2
P[2]	`^AAAA`	increase offset by 1
P[1]	`^TAAAA`	increase offset by 5

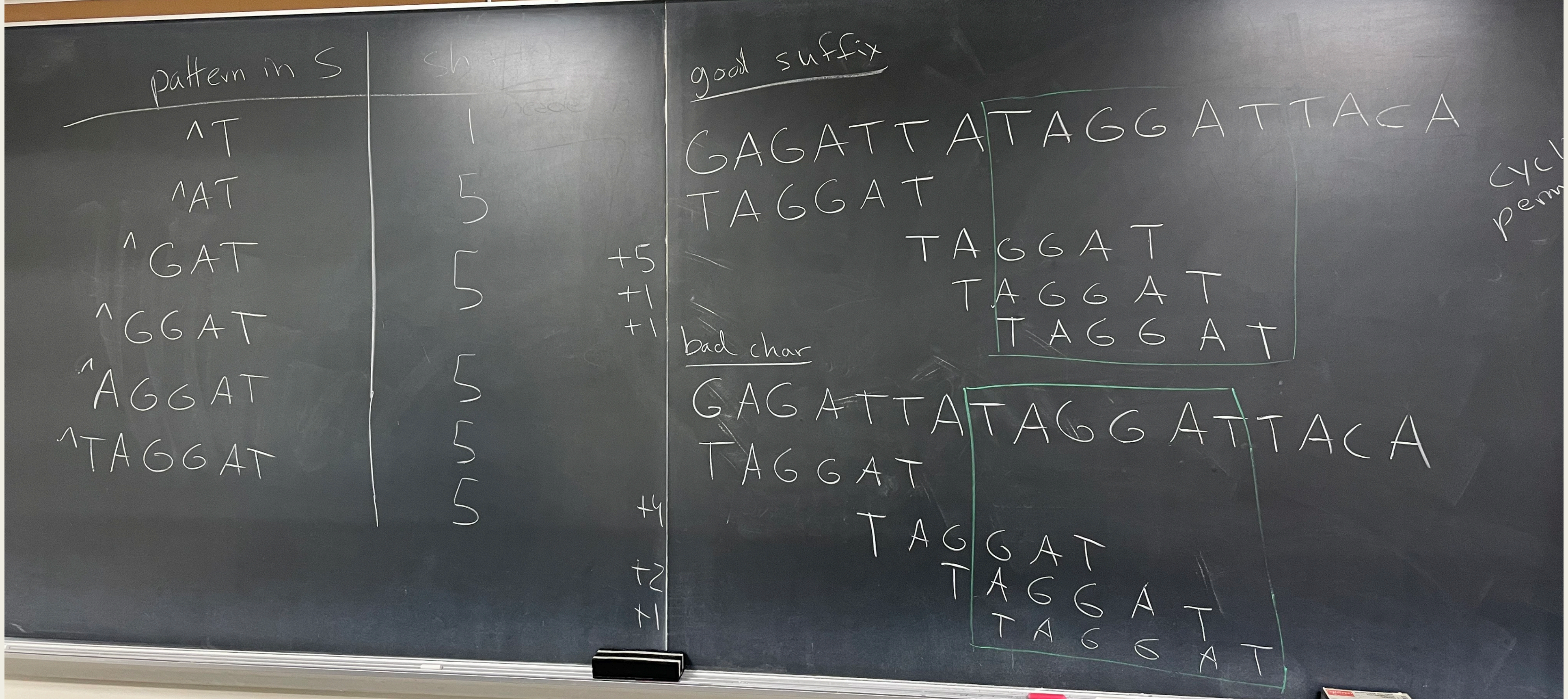
Combine the two rules to complete the Boyer-Moore algorithm

Match P to S, from right to left starting at some offset position of S. If a mismatch is found compute

shift = max(bad character rule, good suffix rule)

Increase offset by *shift* and start the matching again from the right most position of P.

Good suffix rule worksheet



```
 pattern in S  | sh
       ^T      |  1
      ^AT      |  5
     ^GAT      |  5        +5
    ^GGAT      |           +1
                           +1
   ^AGGAT      |  5
  ^TAGGAT      |  5
               |  5        +4

                           +2
                           +1
```

good suffix

```
GAGATTATAGGATTACA
TAGGAT
       TAGGAT
        TAGGAT
         TAGGAT
```

bad char

```
GAGATTATAGGATTACA
TAGGAT
     TAGGAT
       TAGGAT
        TAGGAT
```

cycl
perm

Boyer-More Complexity

Best case?

```
S = AAACAAACAAACAAAT
    AAAT
        AAAT
```

Maximum num comparisons: O(N/M)

Gets more efficient as the pattern gets longer!

Average case?

```
S = AGTCTAGCTAGCATCGACTACGAC
    ACGT
      ACGT
         ACGT
```

Average num comparisons: O(N)

Small alphabet

```
S = AGTCTAGCTAGCATCGACTACGAC
    ACGT
      ACGT
         ACGT
```

Large alphabet

```
S = ASDFAJNSDAWZA#XXKXLS#%K
    XZAK
        XZAK
            AZAK
```

Bigger skip with larger alphabet

Worst case?

```
S = AAAAAAAAAAAAAAAA
    AAAT
     AAAT
```

No skip: O(N)

Worst case?

```
S:  AAAAAAAAAAAAAAAAAAAA
P:  AAAAA
```

No skip, no mismatch: O(NM)

* Worst case is O(N) if the pattern does not appear in the text

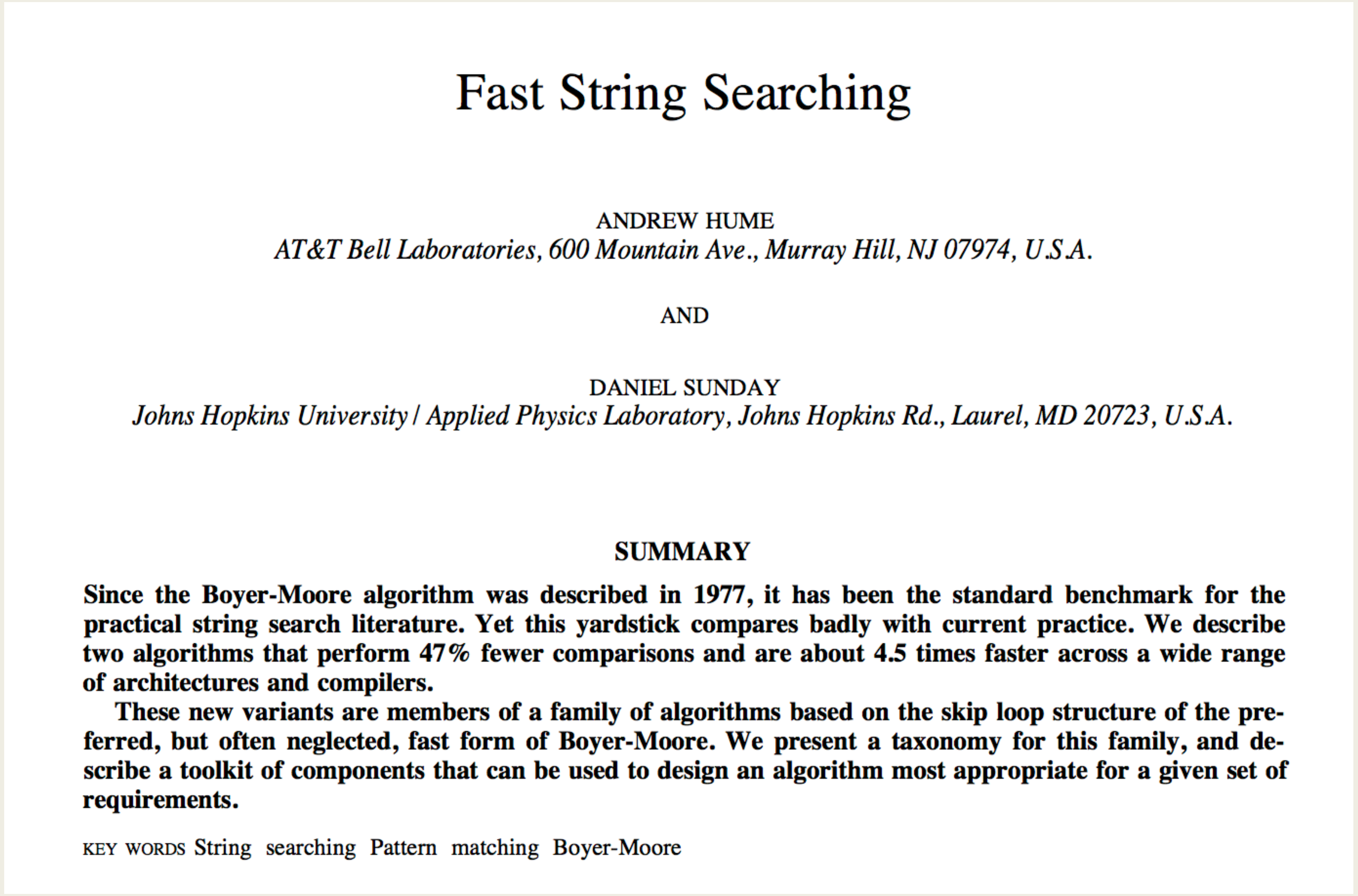
Remarks on Boyer-Moore

- O(N/M) -> Becomes more efficient, the longer the pattern
- Small memory complexity.
- Not as efficient when we have a small alphabet size

Remarks on Boyer-Moore

- Boyer-Moore and variations are good general search algorithms
- At the relatively modest cost of O(M) preprocessing, they produce substantial speed-ups over the naïve algorithm
- This is hard to see abstractly, but in practice they have very good performance (i.e. the average case is close to the best case)
- When you hit Ctrl-F in a website or text document, it’s running a variation of Boyer-Moore under the hood

Extra reading



Fast String Searching

ANDREW HUME
AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974, U.S.A.

AND

DANIEL SUNDAY
Johns Hopkins University / Applied Physics Laboratory, Johns Hopkins Rd., Laurel, MD 20723, U.S.A.

SUMMARY

Since the Boyer-Moore algorithm was described in 1977, it has been the standard benchmark for the practical string search literature. Yet this yardstick compares badly with current practice. We describe two algorithms that perform 47% fewer comparisons and are about 4.5 times faster across a wide range of architectures and compilers.

These new variants are members of a family of algorithms based on the skip loop structure of the preferred, but often neglected, fast form of Boyer-Moore. We present a taxonomy for this family, and describe a toolkit of components that can be used to design an algorithm most appropriate for a given set of requirements.

KEY WORDS String searching Pattern matching Boyer-Moore

For optimal performance we start to care about 1) constants 2) characteristics of the data

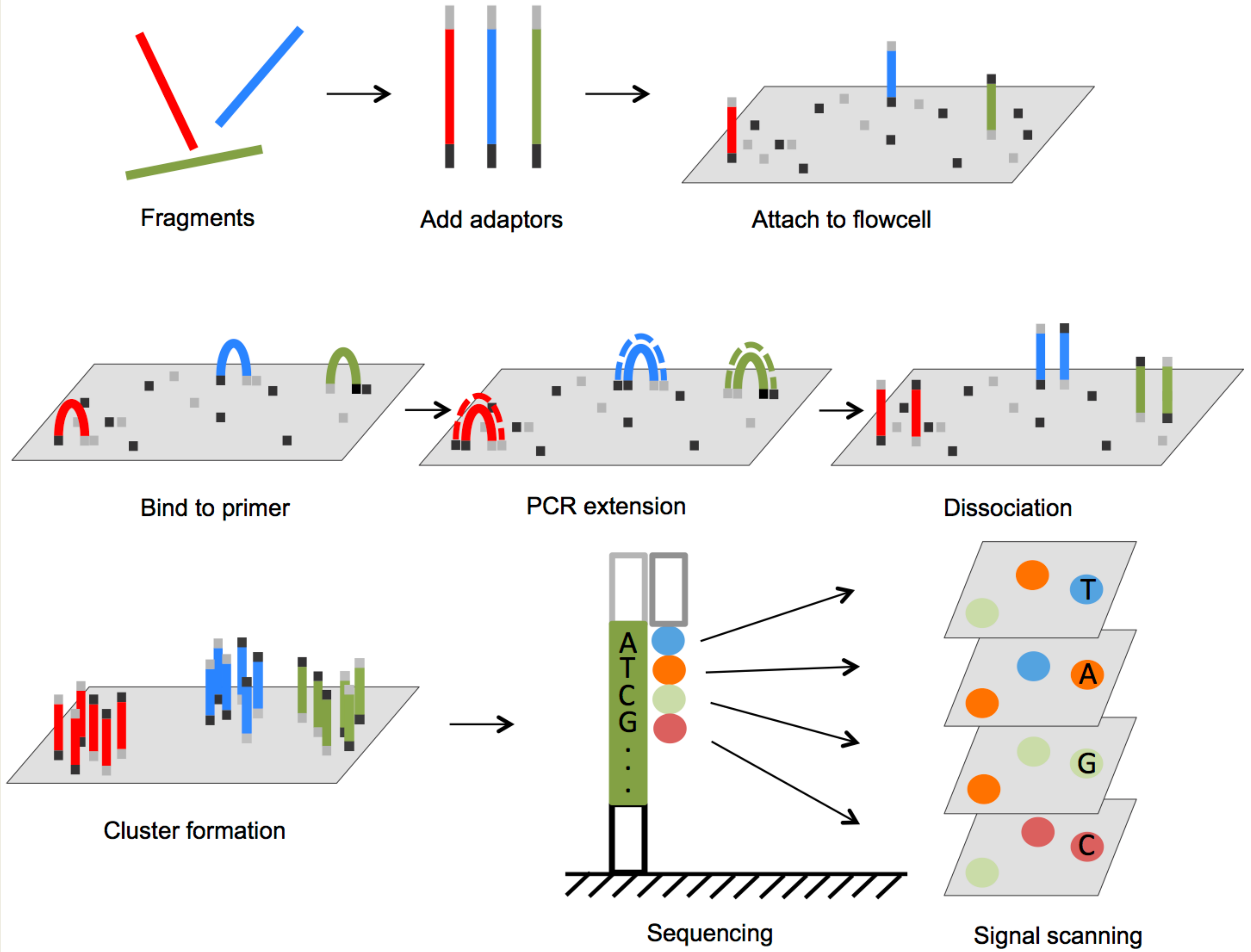
Final thought…

- We sped up our algorithm by doing some pre-processing of the pattern

- Often (e.g. read mapping), we want to match a large number of patterns (reads) to a search string (reference genome) that doesn’t change

- Is there some way we could speed this up by pre-processing the search string?

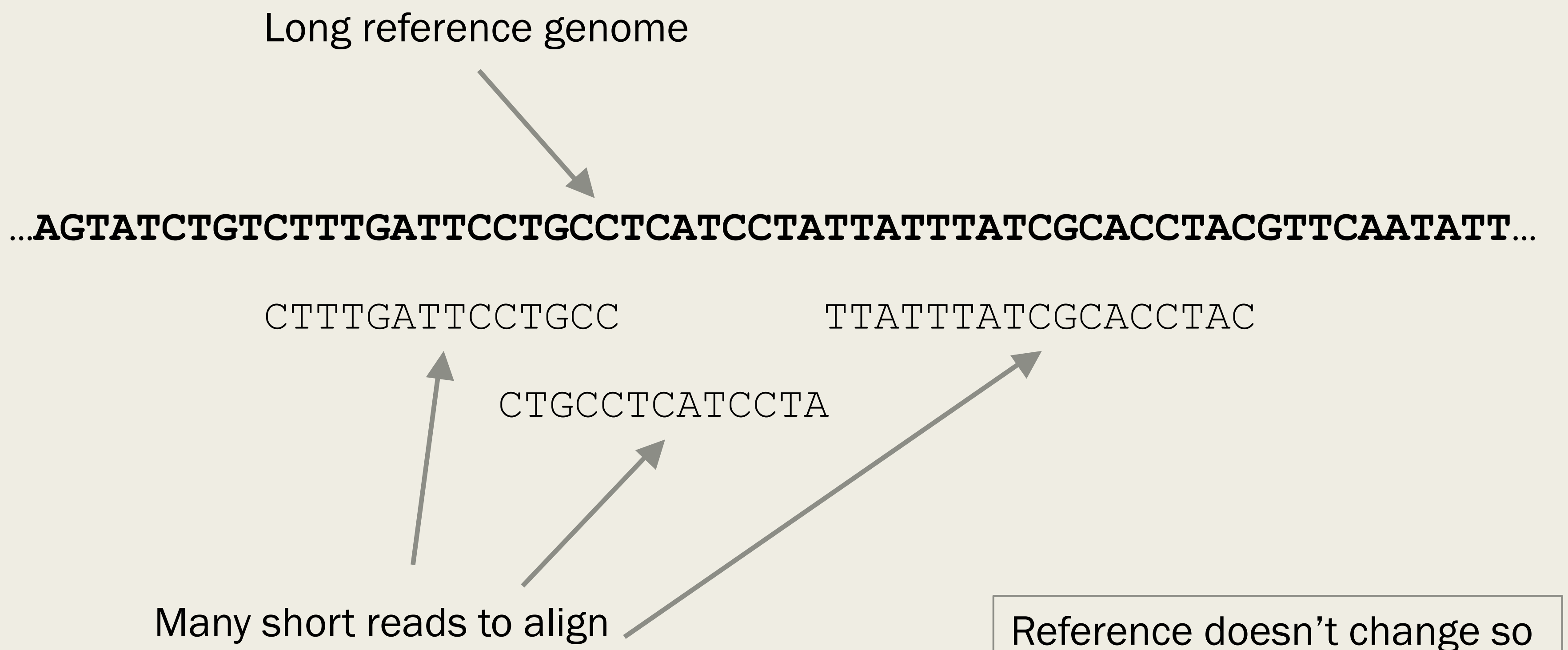
Sequencing Pipeline



InTech open science

Read Mapping

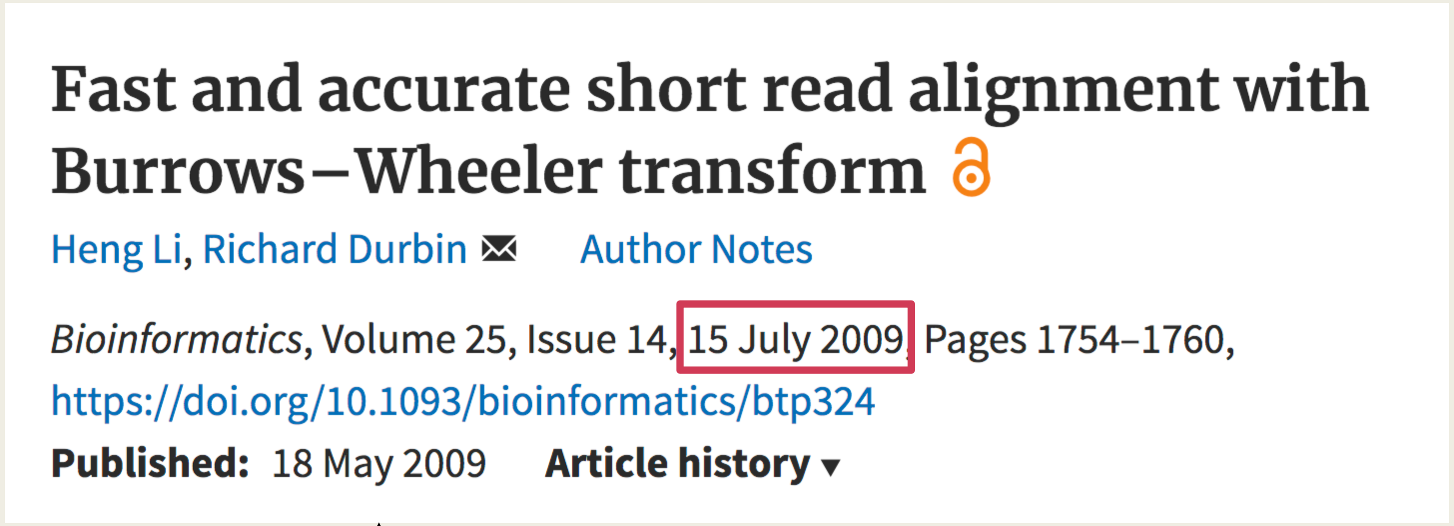
Read mapping


Long reference genome
…AGTATCTGTCTTTGATTCCTGCCTCATCCTATTATTTATCGCACCTACGTTCAATATT…
CTTTGATTCCTGCC
TTATTTATCGCACCTAC
CTGCCTCATCCTA
Many short reads to align

Reference doesn’t change so lets process it once (slow) but then hopefully it will be fast to map each read.

Bowtie and BWA (posted reading)

- First practical read aligners to use the Burrows-Wheeler Transform



Fast and accurate short read alignment with Burrows–Wheeler transform

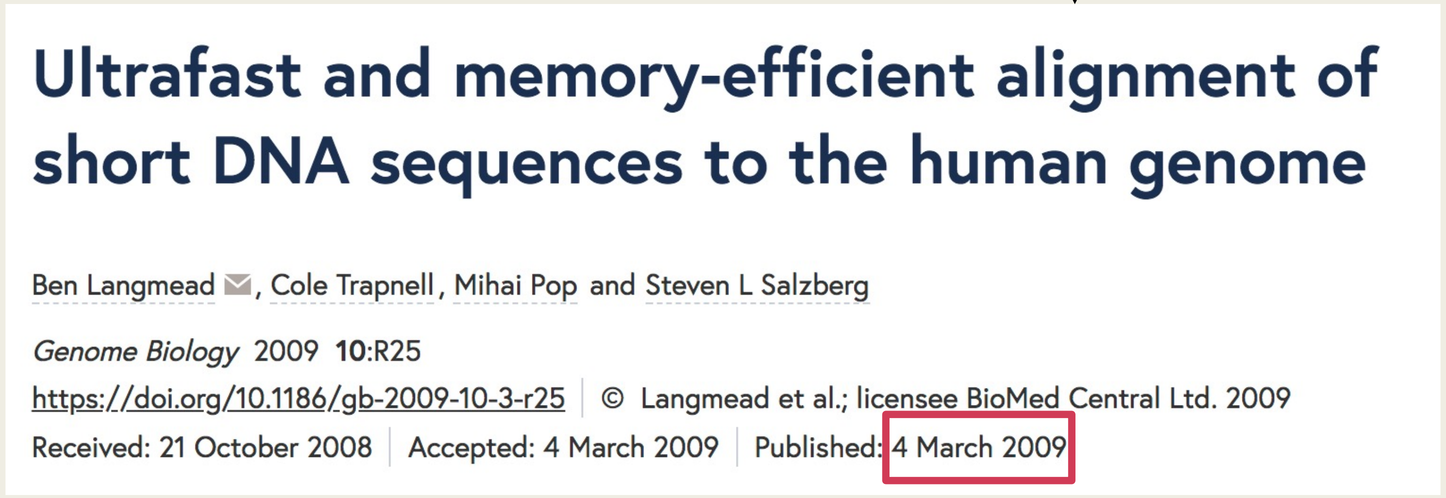
Heng Li, Richard Durbin   Author Notes

Bioinformatics, Volume 25, Issue 14, 15 July 2009, Pages 1754–1760, https://doi.org/10.1093/bioinformatics/btp324

Published: 18 May 2009   **Article history** ▾

BWA

Bowtie



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

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

Genome Biology 2009 **10**:R25

https://doi.org/10.1186/gb-2009-10-3-r25 | © Langmead et al.; licensee BioMed Central Ltd. 2009

Received: 21 October 2008 | Accepted: 4 March 2009 | Published: 4 March 2009

Comparison of Bowtie and BWA

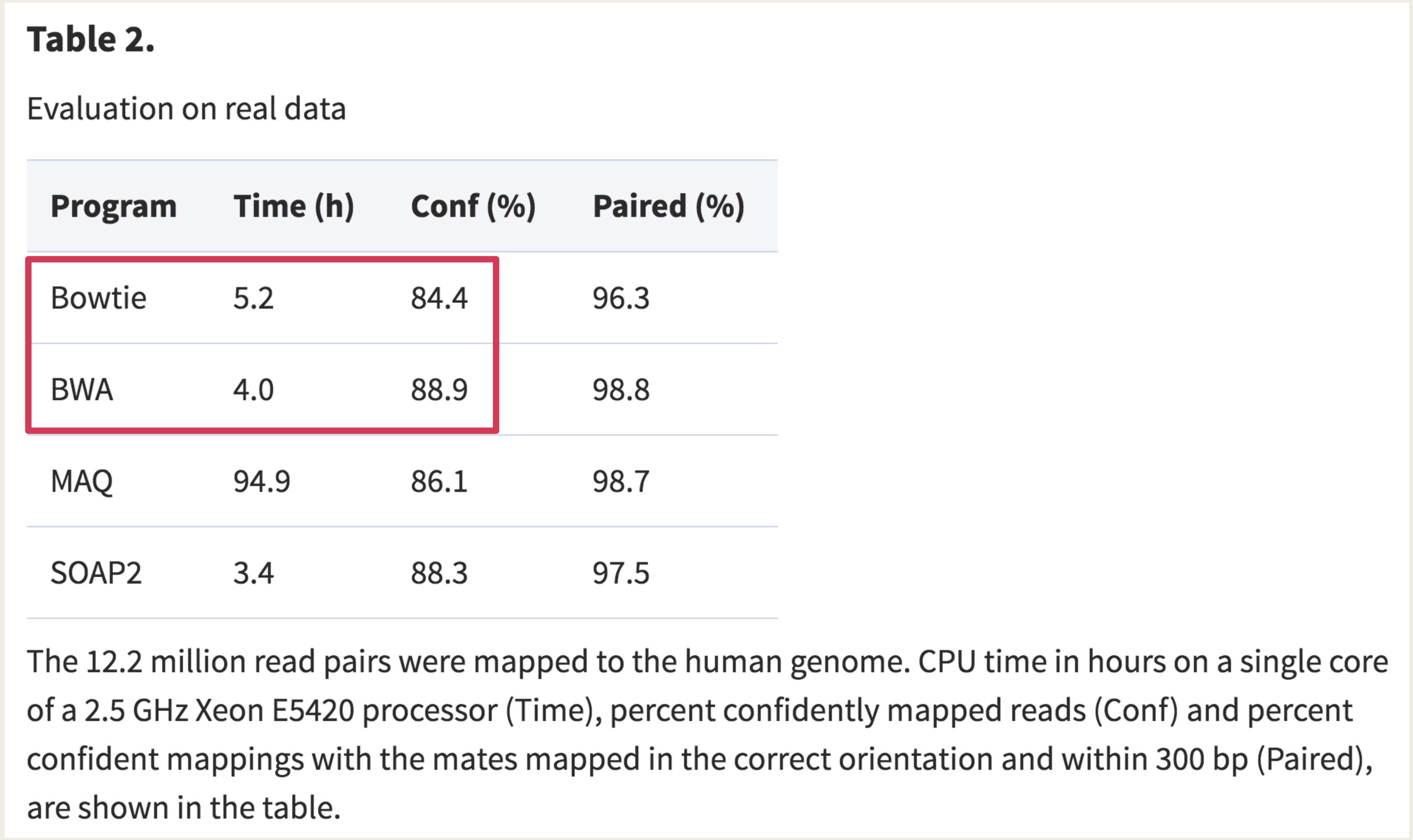


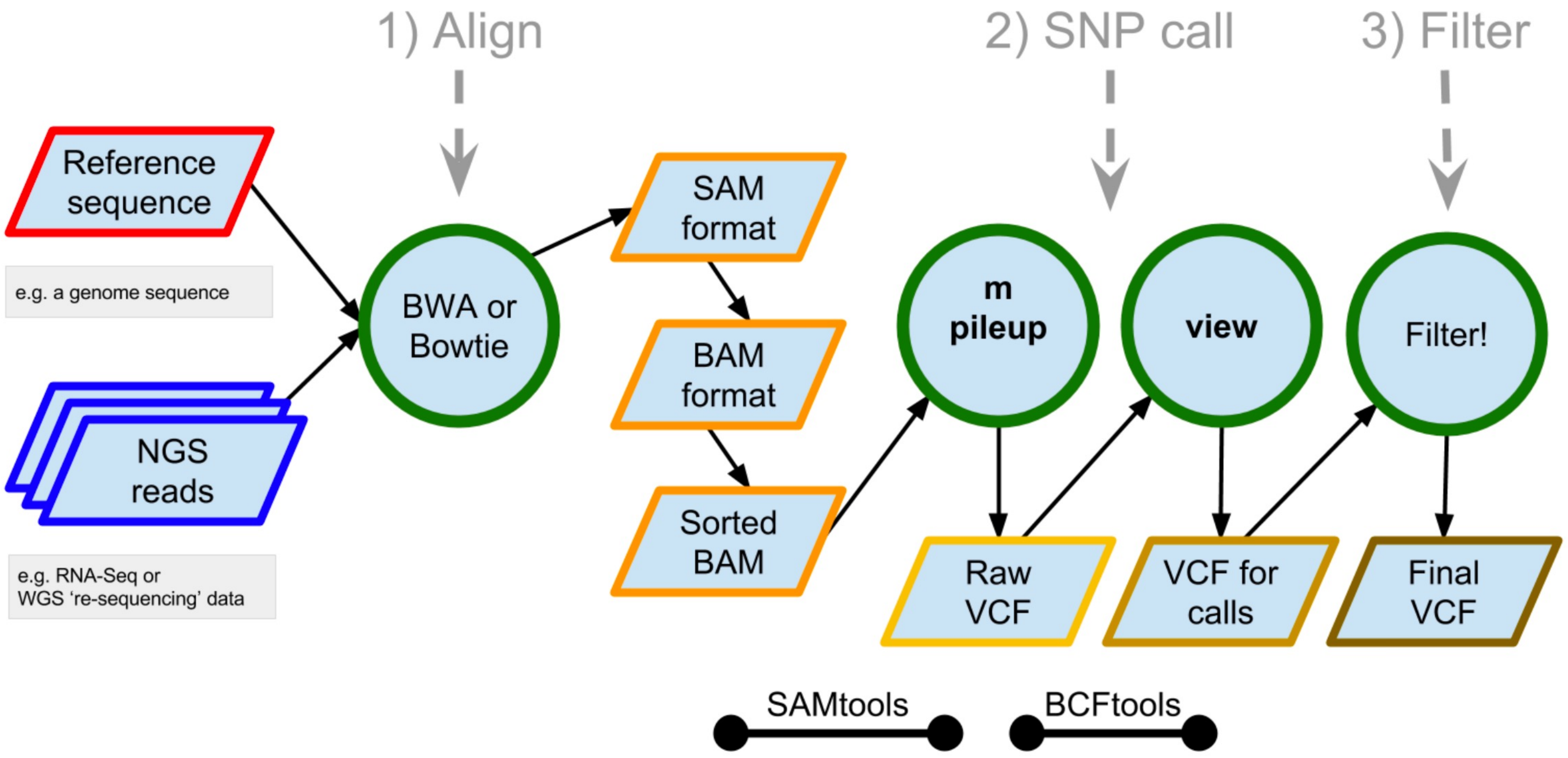
Table 2.

Evaluation on real data

Program	Time (h)	Conf (%)	Paired (%)
Bowtie	5.2	84.4	96.3
BWA	4.0	88.9	98.8
MAQ	94.9	86.1	98.7
SOAP2	3.4	88.3	97.5

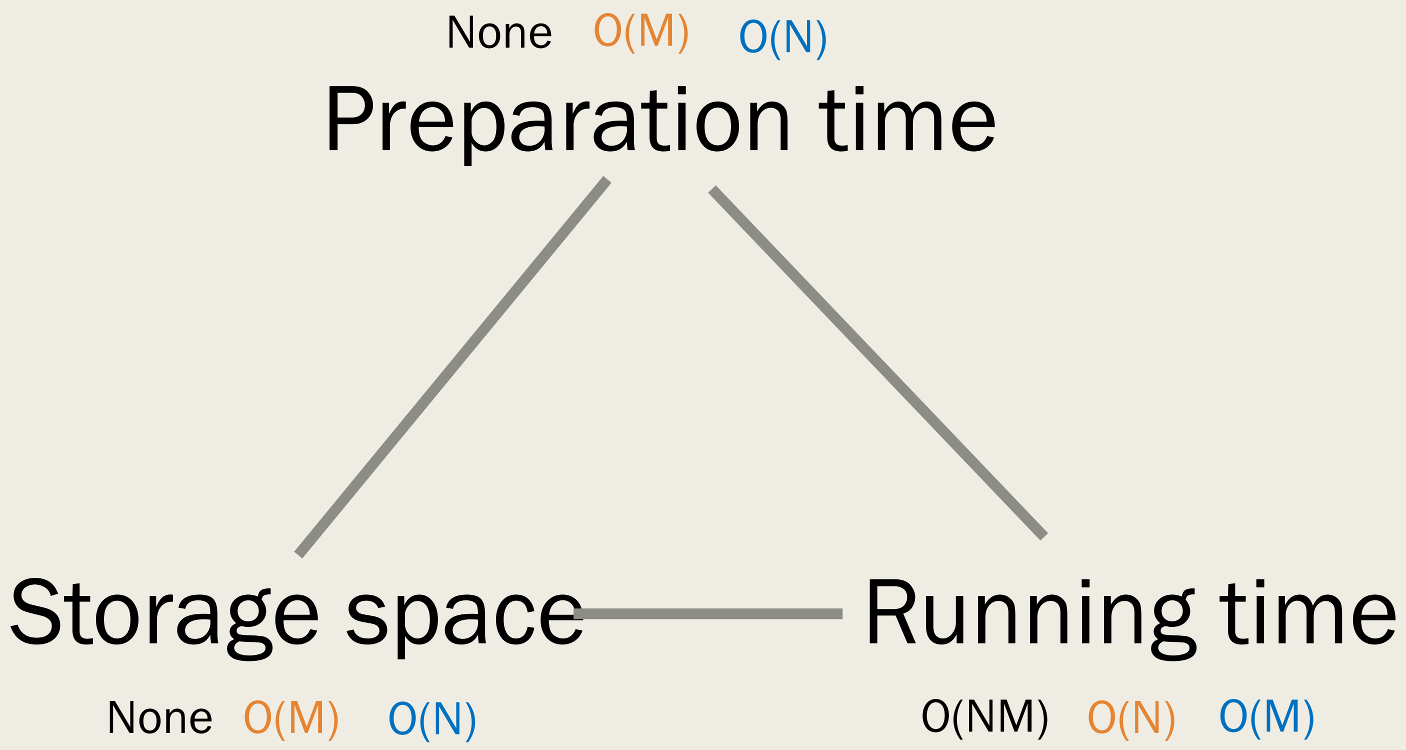
The 12.2 million read pairs were mapped to the human genome. CPU time in hours on a single core of a 2.5 GHz Xeon E5420 processor (Time), percent confidently mapped reads (Conf) and percent confident mappings with the mates mapped in the correct orientation and within 300 bp (Paired), are shown in the table.

Pipeline overview



Credit: Dan Bolster (EBI)

No such thing as a free lunch



Naïve search

Boyer-Moore and variations

Hash tables and BWT

k-mer hashing

```
AGTATCTGTCTTTGATTCCTGCCTCATCCTATTATTTATCGCACCTACGTTCAATATT
```

```
AGTA: 1
```

Recall: k-mer is a string of length k. We'll write things like 4-mer, 32-mer etc… to refer to specific lengths

k-mer hashing

```
AGTATCTGTCTTTGATTCCTGCCTCATCCTATTATTTATCGCACCTACGTTCAATATT
```

```
AGTA:  1
GTAT:  2
```

k-mer hashing

AG$\boxed{\text{TATC}}$TGTCTTTGATTCCTGCCTCATCCTATTATTTATCGCACCTACGTTCAATATT

AGTA:	1
GTAT:	2
TATC:	3

k-mer hashing

AGTATCTGTCTGTGATTCCTGCCTCATCCTATTATTTATCGCACCTCTGTTCAATATT

AGTA:	1
GTAT:	2
TATC:	3
ATCT:	4
TCTG:	5,9,42
CTGT:	6
TGTC:	7
GTCT:	8
TCTT:	9

.

.

.

.

.

Collision!

k-mer hashing

```
AGTATCTGTCTGTGATTCCTGCCTCATCCTATTATTTATCGCACCTCTGTTCAATATT
```

AGTA:	1
GTAT:	2
TATC:	3
ATCT:	4
TCTG:	5,9,42
CTGT:	6
TGTC:	7
GTCT:	8
TCTT:	9
.	
.	
.	
.	
.	

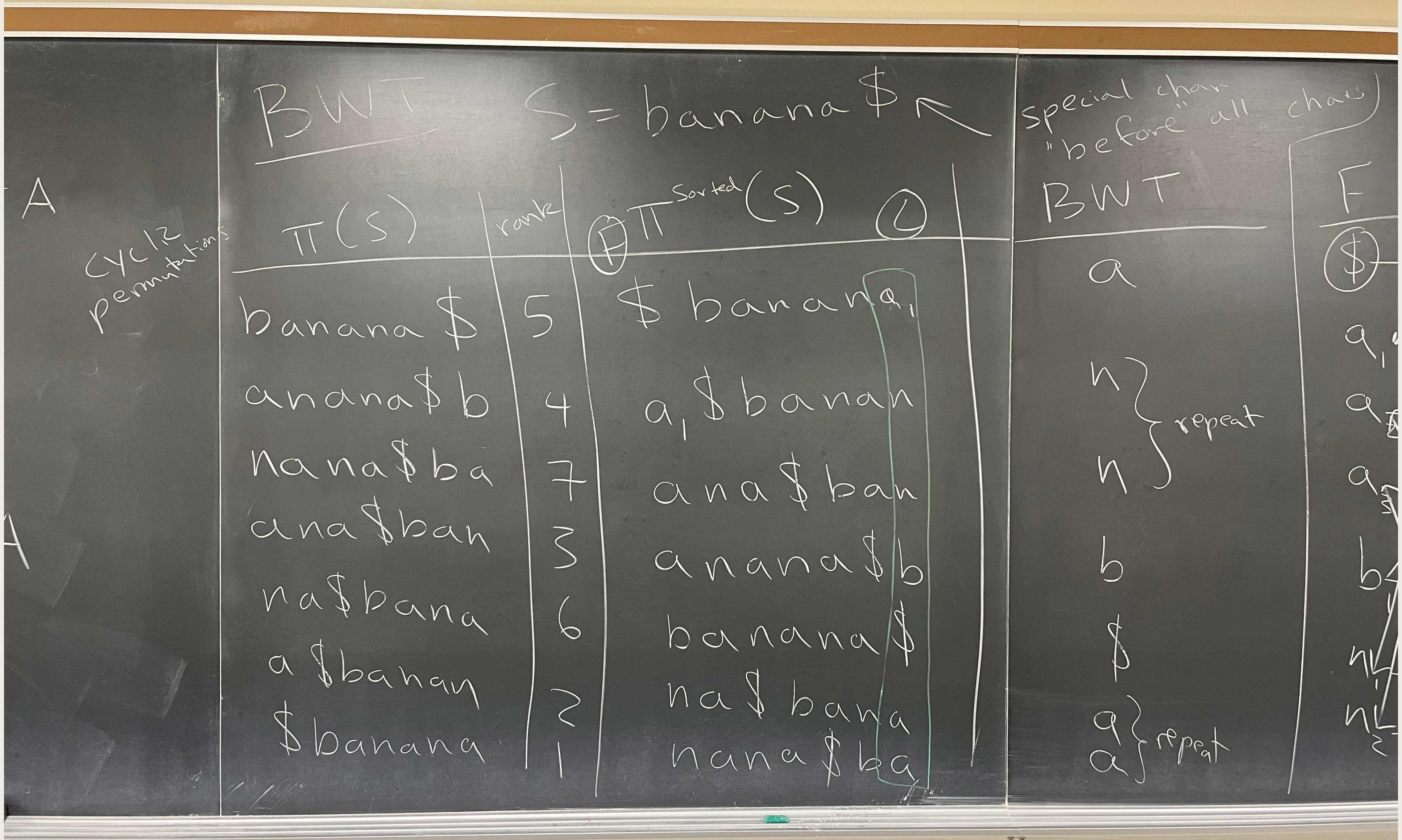
Now, suppose I want to look up:

```
GTCTGTGATTCC
```

1. Take first k-mer in pattern
2. Look up positions in index
3. Check each position for match

How many different k-mers are there?

Burrows-Wheeler Transform (BWT)



A

A

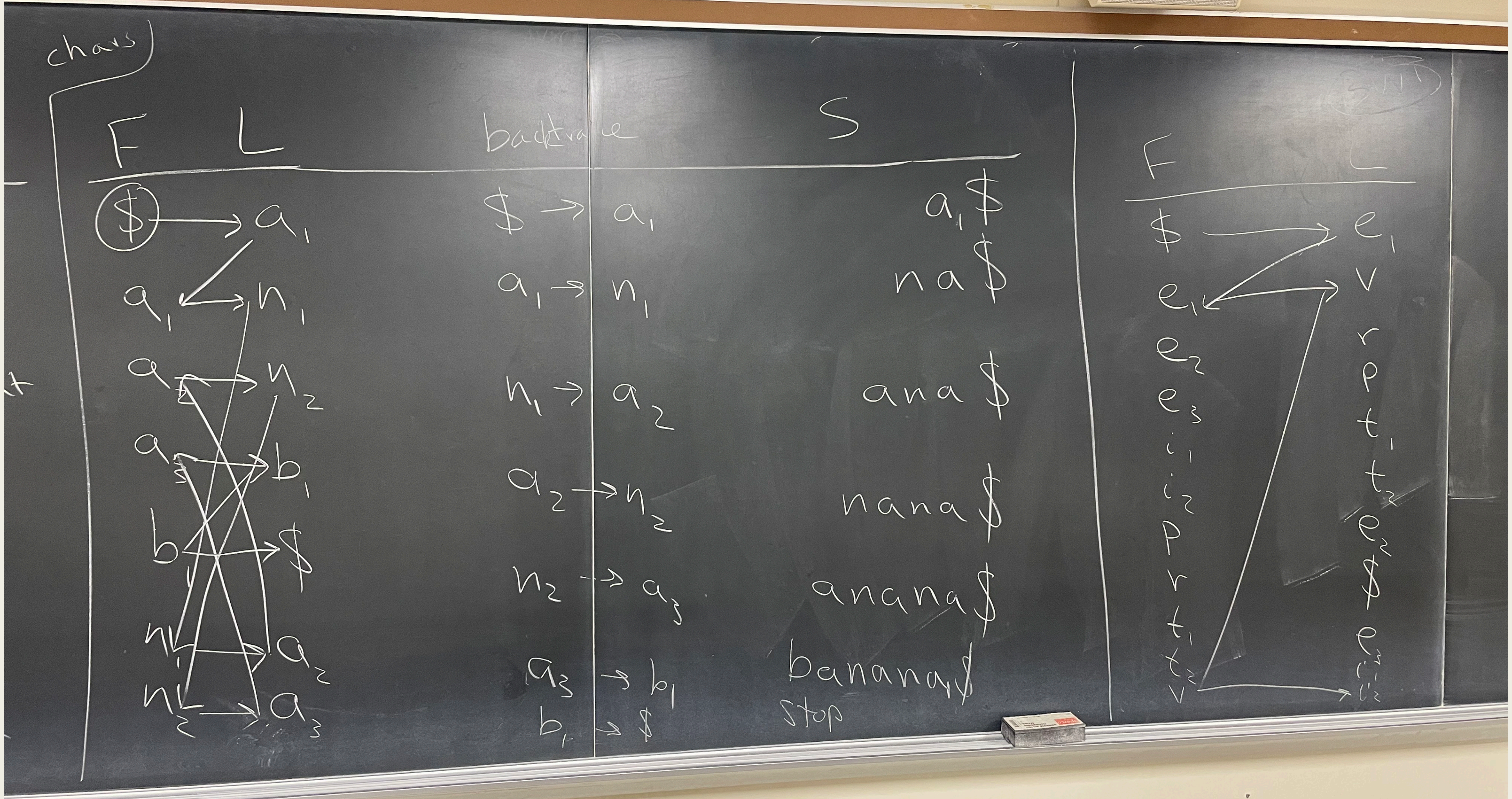
BWT $S = \text{banana\$}$ ↖ special char "before" all chars

cycle permutations

$\pi(S)$	rank	$\boxed{F}\ \pi^{\text{sorted}}(S)\ \boxed{L}$	BWT	F
banana\$	5	\$ banana_1	a	$\boxed{\$}$
anana\$b	4	a_1\$ banan	n	a_1
nana\$ba	7	ana\$ban	n	a
ana\$ban	3	anana\$b	b	a
na\$bana	6	banana\$	\$	b
a\$banan	2	na\$bana	a	n
\$banana	1	nana\$ba	a	n

$\left.\begin{matrix}n\\n\end{matrix}\right\}$ repeat

$\left.\begin{matrix}a\\a\end{matrix}\right\}$ repeat



chars

F	L	backtrace		S
$	a_1	$ \$ \rightarrow a_1 $		$a_1\$$
a_1	n_1	$a_1 \rightarrow n_1$		na$
a_2	n_2	$n_1 \rightarrow a_2$		ana$
a_3	b_1	$a_2 \rightarrow n_2$		nana$
b_1	$	$n_2 \rightarrow a_3$		anana$
n_1	a_2	$a_3 \rightarrow b_1$		banana$
n_2	a_3	$b_1 \rightarrow \$$		stop

BWT

F	L
$	e_1
e_1	v
e_2	r
e_3	p
i_1	t_1
i_2	t_2
p	e_2
r	$
t_1	e_3
t_2	i_1
v	i_2