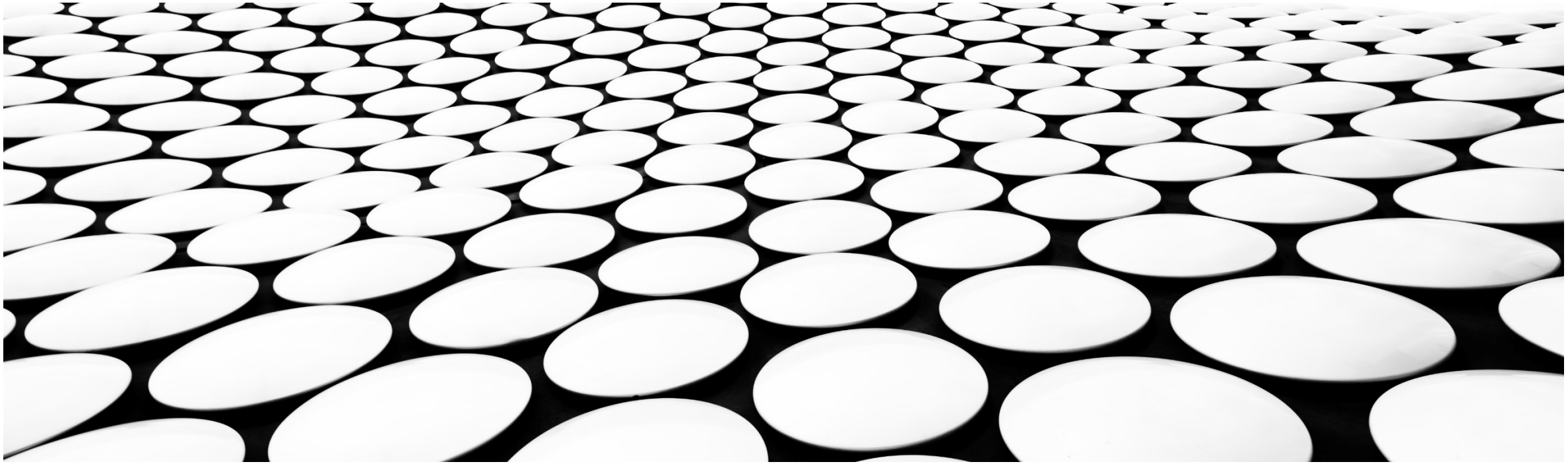
BIOINFORMATICS(BIOCOMPUTING)

(4)

BOYER-MOORE

DR. IBRAHIM ZAGHLOUL



String Definitions

A *string* S is a finite ordered list of characters.

Characters are drawn from an alphabet Σ.

Nucleic acid alphabet: { A, C, G, T }
Amino acid alphabet: { A, R, N, D, C, E, Q, G, H, I, L, K, M, F, P, S, T, W, Y, V }

Length of S, $|S|$, is the number of characters in S

ϵ is the empty string. $|\epsilon| = 0$

String Definitions

- For strings *S* and *T* over *Σ*, their *concatenation* consists of the characters of *S* followed by the characters of *T*, denoted *ST*

- S is a substring of T if there exist (possibly empty) strings u and v such that T = uSv

- S is a prefix of T if there exists a string u such that T = Su.
  If neither S nor u are ϵ, S is a proper prefix of T.
- Definitions of suffix and proper suffix are similar.

String Definitions

- We defined substring. Subsequence is similar except the characters need not be consecutive.

- “cat” is a substring and a subsequence of “con**cat**enate”

- “cant” is a subsequence of “con**ca**te**n**a**t**e”, but not a substring

Exact matching

- Looking for places where a *pattern P* occurs as a substring of a *text T*. Each such place is an *occurrence* or *match*.

- An *alignment* is a way of putting *P's* characters opposite *T's* characters. It may or may not correspond to an occurrence.

At what offsets does *pattern P* occur within *text T*?

```
P: word
T: There would have been a time for such a word
```

Answer: offset 40

Exact Matching

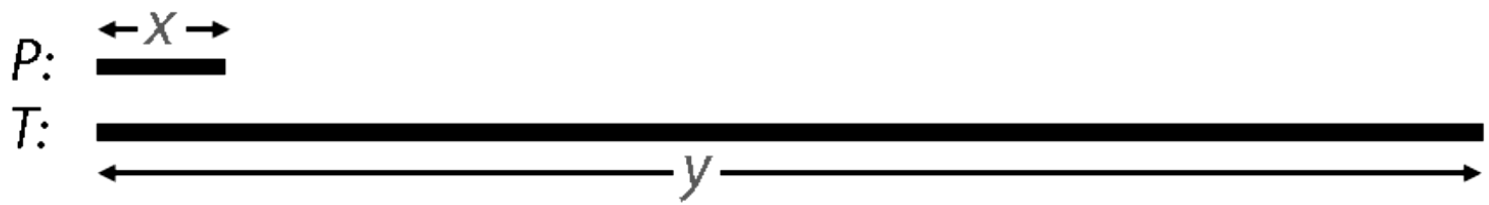
At what offsets does *pattern P* occur within *text T*?

```
>>> t = 'There would have been a time for such a word'
>>> t.find('word')
40
```

Exact matching: naïve algorithm

Let $x = |P|, y = |T|$

How many alignments are possible given x and y?



$$y - x + 1$$

Exact matching: naïve algorithm

Let $x = |P|, y = |T|$

What's the greatest # character comparisons possible?

P: aaaa

T: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

```
aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
  aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
   aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
    aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
```

$x(y - x + 1)$

Exact matching: naïve algorithm

Let $x = |P|, y = |T|$

What's the **least** # character comparisons possible?

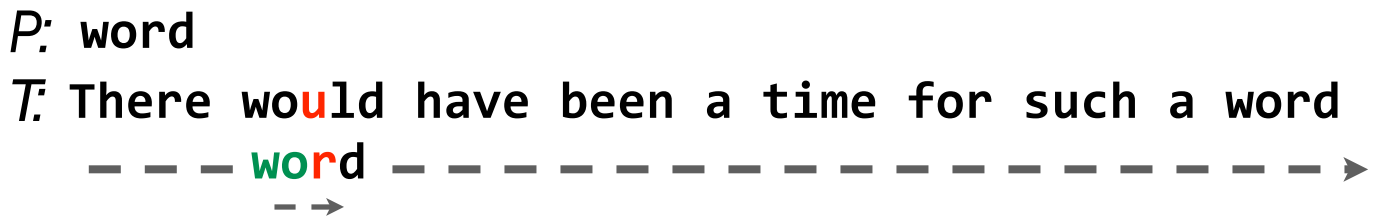
P: abbb

T: bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb

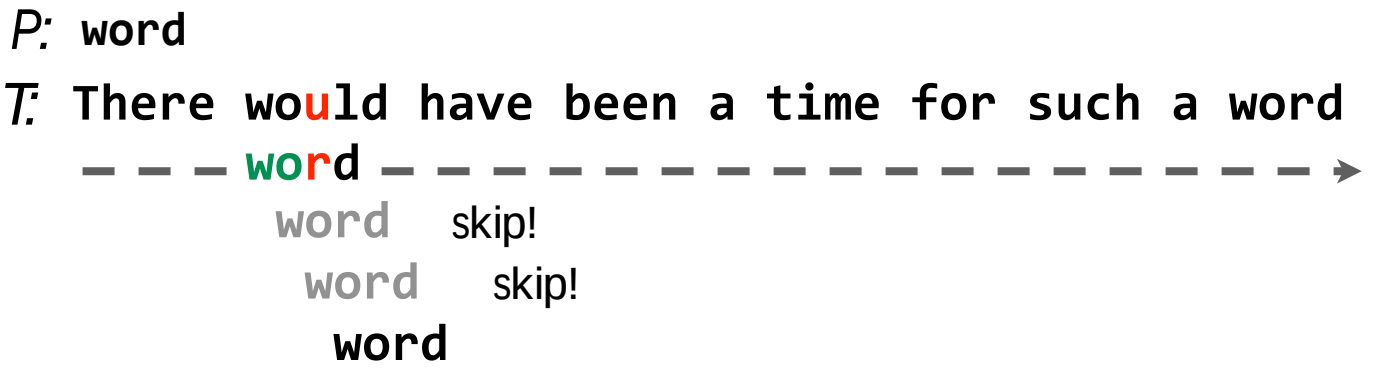
abbb abbb abbb abbb abbb abbb abbb abbb abbb
abbb abbb abbb abbb abbb abbb abbb abbb
abbb abbb abbb abbb abbb abbb abbb abbb
abbb abbb abbb abbb abbb abbb abbb abbb
abbb abbb abbb abbb abbb abbb abbb abbb

$y - x + 1$

Can we improve on the naïve algorithm?



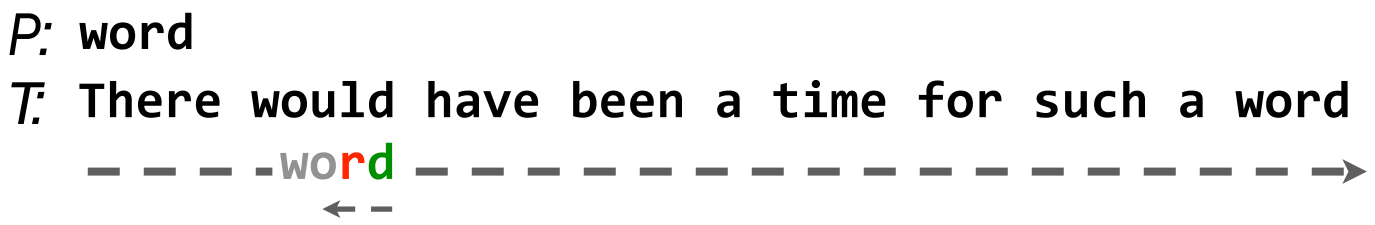
u doesn't occur in *P*, so skip next two alignments



Boyer-Moore

Learn from character comparisons to skip pointless alignments

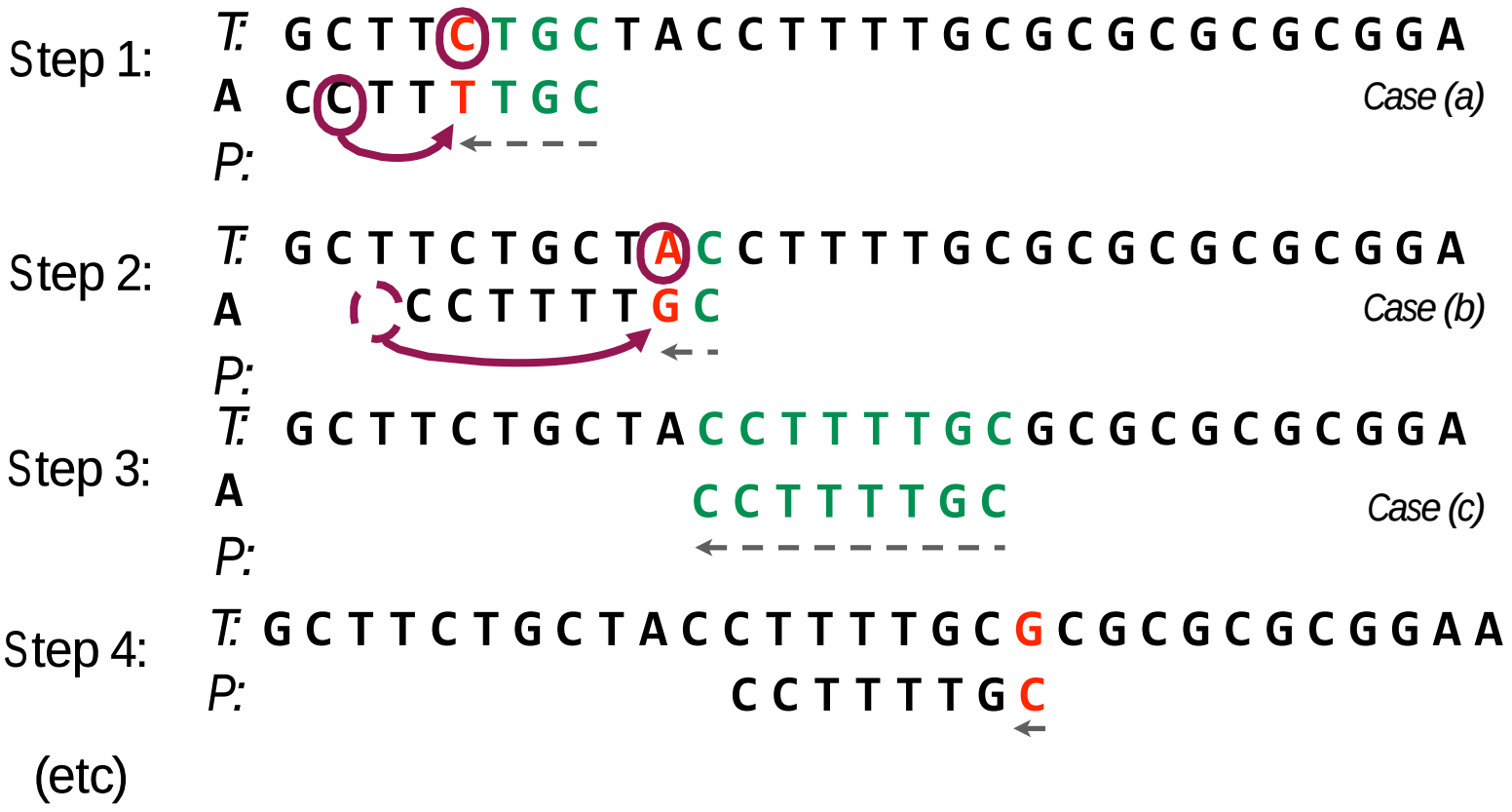
1. When we hit a mismatch, move *P* along until the mismatch becomes a match — "Bad character rule"
2. When we move *P* along, make sure characters that matched in the last alignment also match in the next alignment — "Good suffix rule"
3. Try alignments in one direction, but do character comparisons in *opposite* direction — For longer skips



Boyer, RS and Moore, JS. "A fast string searching algorithm." *Communications of the ACM* 20.10 (1977): 762-772.

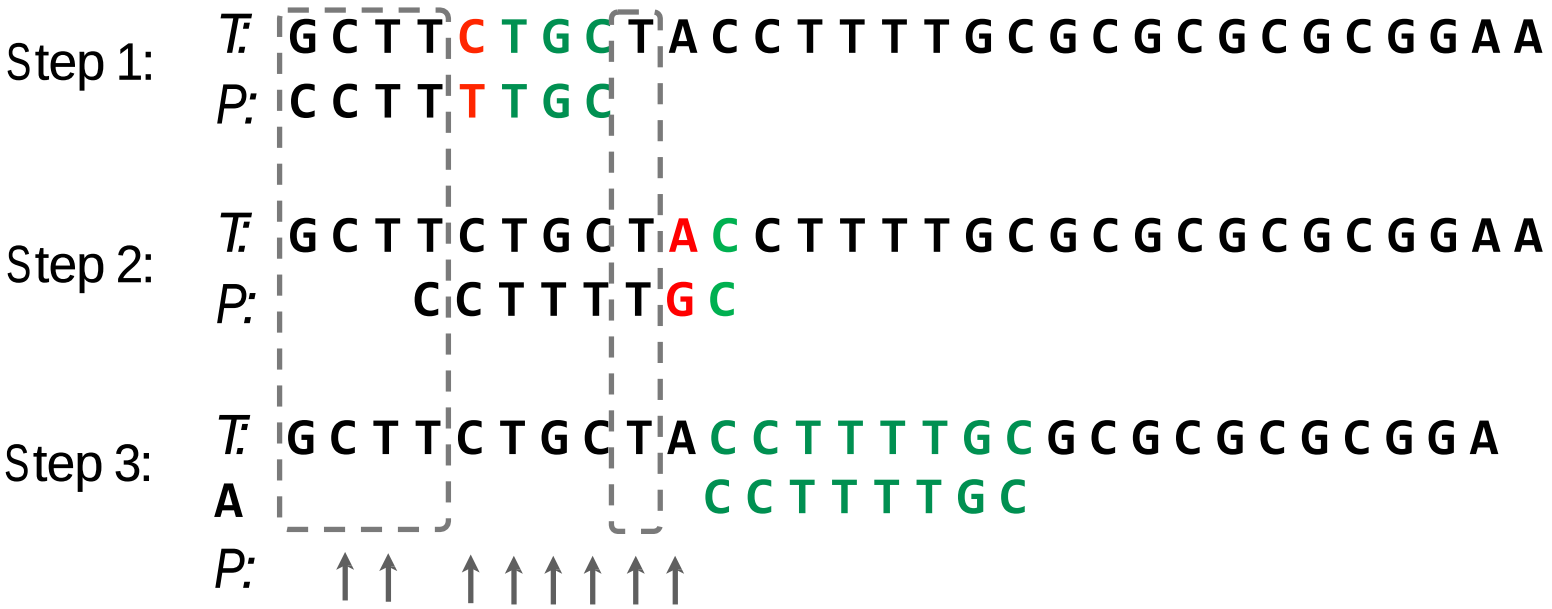
Boyer-Moore: Bad character rule

Upon mismatch, skip alignments until:
(a) mismatch becomes a match, or
(b) *P* moves past mismatched character.
(c) If there was no mismatch, don't skip



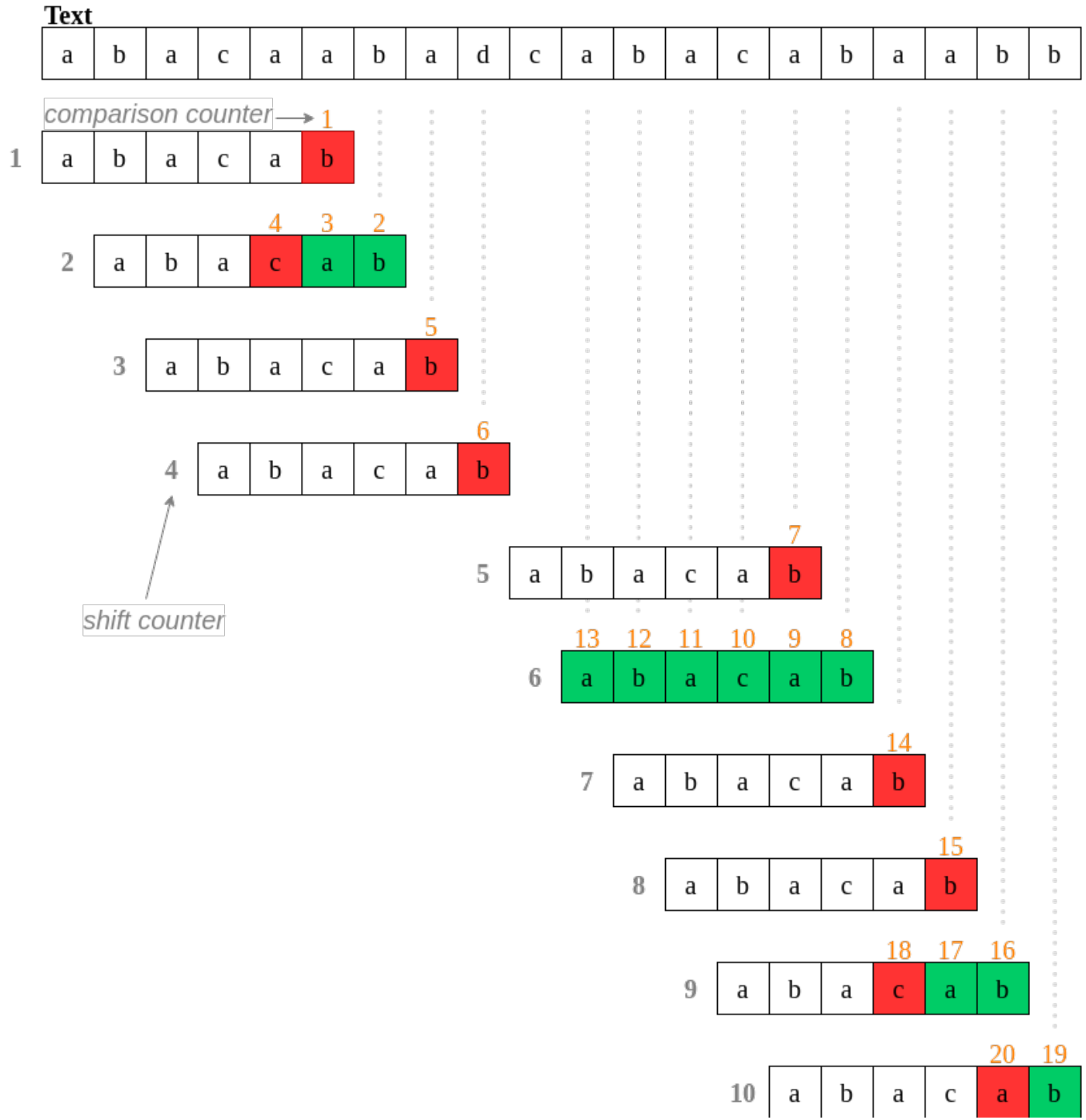
(etc)

Boyer-Moore: Bad character rule



Up to step 3, we skipped 8 alignments

5 characters in *T* were never looked at

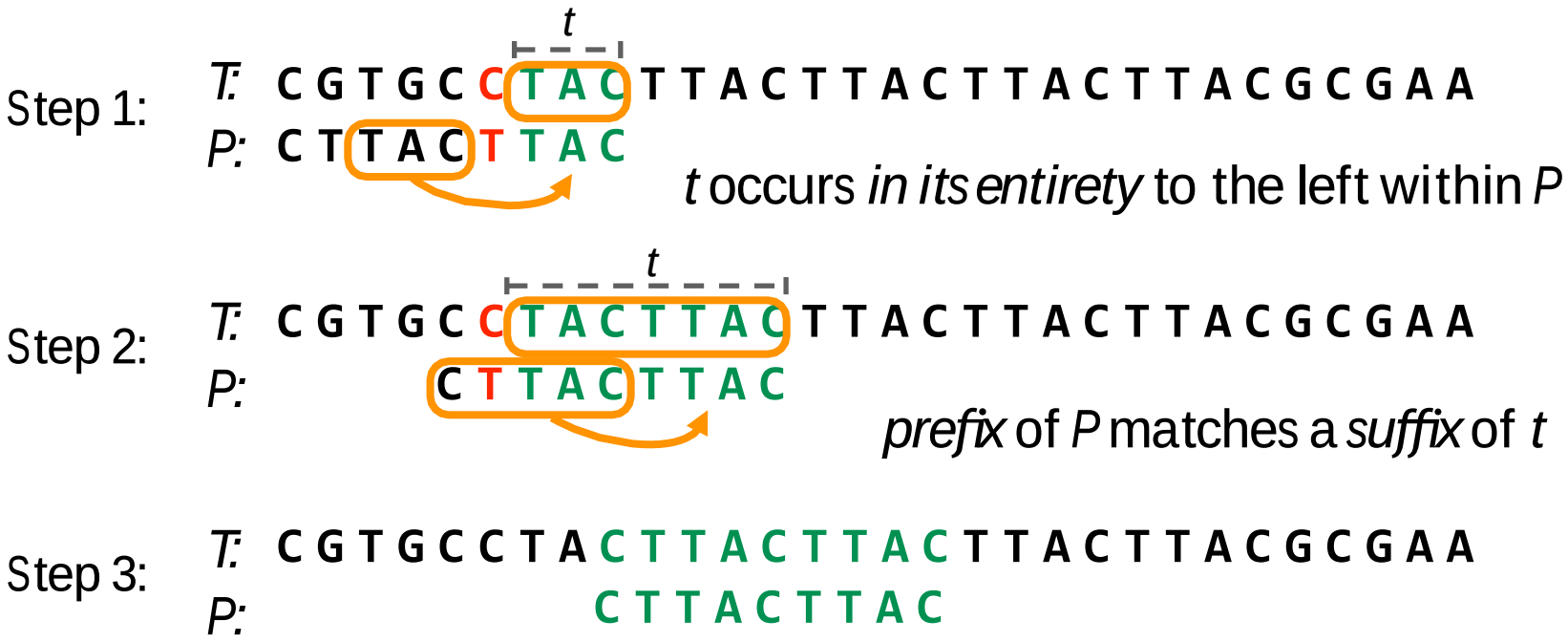

Text
a b a c a a b a d c a b a c a b a a b b
comparison counter
1
1 a b a c a b
4 3 2
2 a b a c a b
5
3 a b a c a b
6
4 a b a c a b
shift counter
7
5 a b a c a b
13 12 11 10 9 8
6 a b a c a b
14
7 a b a c a b
15
8 a b a c a b
18 17 16
9 a b a c a b
20 19
10 a b a c a b

Boyer-Moore: Good suffix rule

Let *t* = substring matched by inner loop; skip until (a) there are no mismatches between *P* and *t or* (b) *P* moves past *t*

Boyer-Moore: Good suffix rule

Let *t* = substring matched by inner loop; skip until (a) there are no mismatches between *P* and *t or* (b) *P* moves past *t*



Case (a) has two subcases according to whether *t* occurs *in its entirety* to the left within *P* (as in step 1), or a *prefix* of *P* matches a *suffix* of *t* (as in step 2)

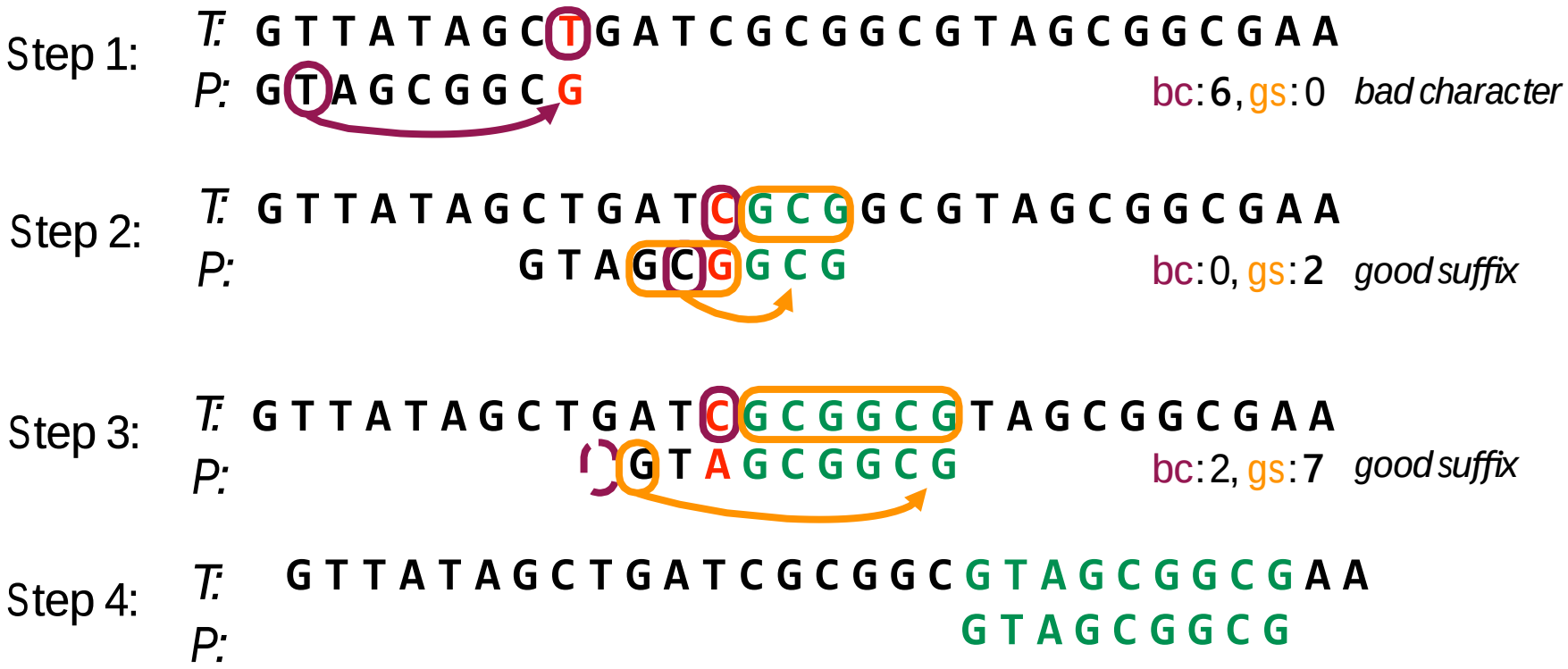
Boyer-Moore: Putting it together

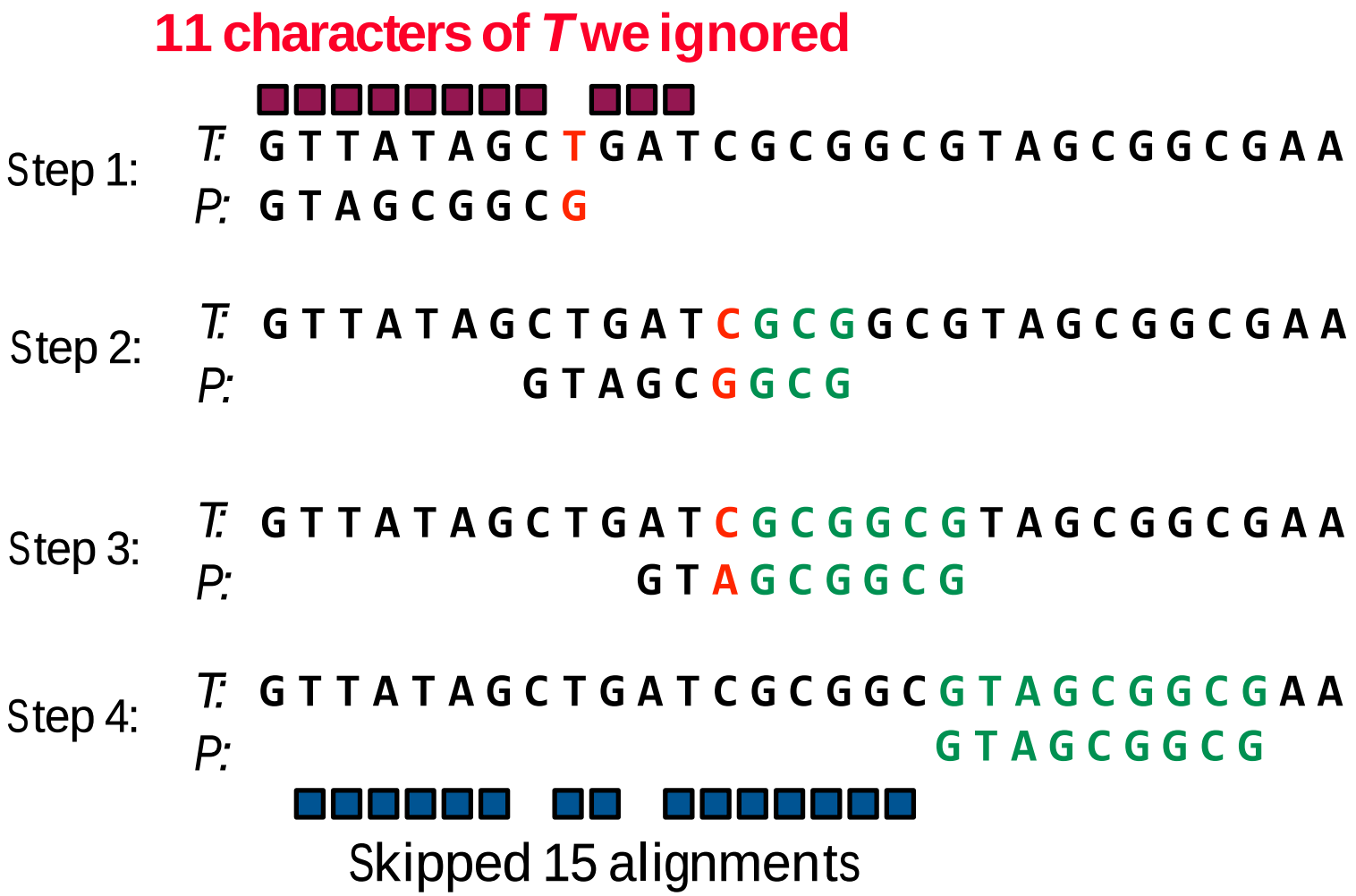
How to *combine* bad character and good suffix rules?

bad char says skip 2, good suffix says skip 7 Take the maximum! (7)

Boyer-Moore: Putting it together

Use bad character or good suffix rule, *whichever skips more*




11 characters of T we ignored
Step 1:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G
Step 2:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G
Step 3:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G
Step 4:
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G
Skipped 15 alignments

Boyer-Moore: Preprocessing

Pre-calculate skips for all possible mismatch scenarios!
For bad character rule and P = TCGC:

P

Σ		T	C	G	C
	A				
	C		-		-
	G			-	
	T	-			

Boyer-Moore: Preprocessing

Pre-calculate skips for all possible mismatch scenarios!
For bad character rule and P = TCGC:

P

Σ	T	C	G	C
A	0	1	2	3
C	0	-	0	-
G	0	1	-	0
T	-	0	1	2

Skip: 1 alignments
(2 shifts)

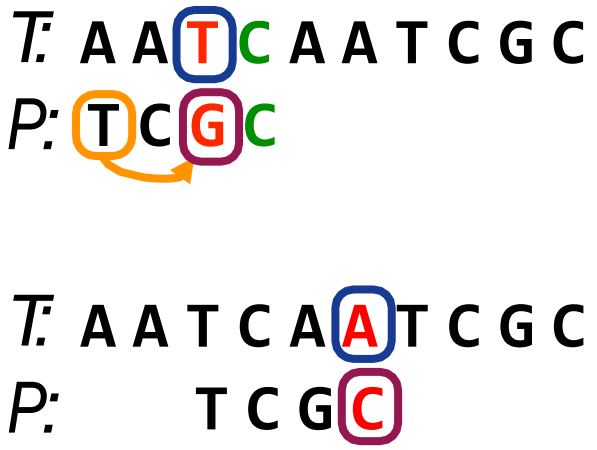
This can be constructed efficiently.

Boyer-Moore: Preprocessing

Pre-calculate skips for all possible mismatch scenarios!
For bad character rule and P = TCGC:

P

Σ	T	C	G	C
A	0	1	2	3
C	0	-	0	-
G	0	1	-	0
T	-	0	1	2



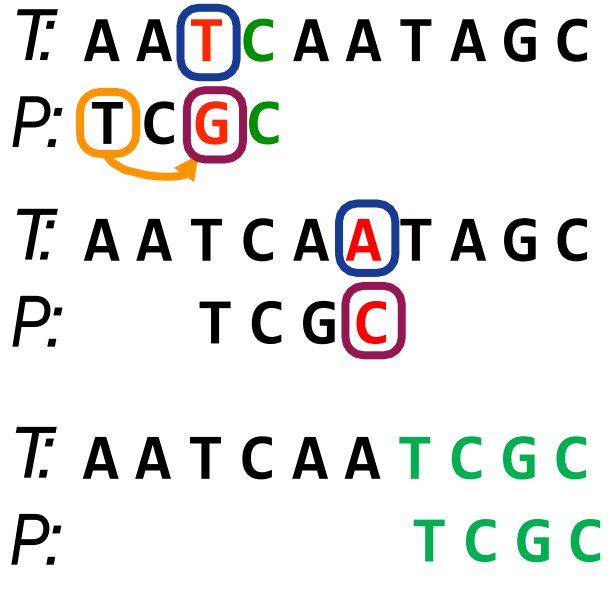
Skip: 3 alignments
(4 shifts)

This can be constructed efficiently.

Boyer-Moore: Preprocessing

Pre-calculate skips for all possible mismatch scenarios!
For bad character rule and P = TCGC:

Σ	P: T	P: C	P: G	P: C
A	0	1	2	3
C	0	-	0	-
G	0	1	-	0
T	-	0	1	2



This can be constructed efficiently.

Boyer-Moore: Preprocessing

Pre-calculate skips for all possible mismatch scenarios!
For bad character rule and P = GTAGCGGCG

P

Σ	G	T	A	G	C	G	G	C	G
A									
C									
G									
T									

P: G T A G C G G C G

T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G

Boyer-Moore: Preprocessing

Pre-calculate skips for all possible mismatch scenarios!
For bad character rule and P = GTAGCGGCG

P

Σ	G	T	A	G	C	G	G	C	G
A			-						
C					-			-	
G	-			-		-	-		-
T		-							

P: G T A G C G G C G

T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G

Boyer-Moore: Preprocessing

Pre-calculate skips for all possible mismatch scenarios!
For bad character rule and *P* = GTAGCGGCG

P

Σ	G	T	A	G	C	G	G	C	G
A	0	1	-	0	1	2	3	4	5
C	0	1	2	3	-	0	1	-	0
G	-	0	1	-	0	-	-	0	-
T	0	-	0	1	2	3	4	5	6

P: G T A G C G G C G

T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G

Boyer-Moore: Preprocessing

Pre-calculate skips for all possible mismatch scenarios!
For bad character rule and *P* = GTAGCGGCG

P

Σ	G	T	A	G	C	G	G	C	G
A	0	1	-	0	1	2	3	4	5
C	0	1	2	3	-	0	1	-	0
G	-	0	1	-	0	-	-	0	-
T	0	-	0	1	2	3	4	5	6

P: G T A G C G G C G

T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G

Boyer-Moore: Preprocessing

Pre-calculate skips for all possible mismatch scenarios!
For bad character rule and *P* = GTAGCGGCG

P

Σ	G	T	A	G	C	G	G	C	G
A	0	1	-	0	1	2	3	4	5
C	0	1	2	3	-	0	1	-	0
G	-	0	1	-	0	-	-	0	-
T	0	-	0	1	2	3	4	5	6

P: G T A G C G G C G

T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G

Boyer-Moore: Preprocessing

Pre-calculate skips for all possible mismatch scenarios!
For bad character rule and *P* = GTAGCGGCG

P

Σ	G	T	A	G	C	G	G	C	G
A	0	1	-	0	1	2	3	4	5
C	0	1	2	3	-	0	1	-	0
G	-	0	1	-	0	-	-	0	-
T	0	-	0	1	2	3	4	5	6

P: G T A G C G G C G

T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G

Boyer-Moore: Preprocessing

Pre-calculate skips for all possible mismatch scenarios!
For bad character rule and *P* = GTAGCGGCG

P

Σ	G	T	A	G	C	G	G	C	G
A	0	1	-	0	1	2	3	4	5
C	0	1	2	3	-	0	1	-	0
G	-	0	1	-	0	-	-	0	-
T	0	-	0	1	2	3	4	5	6

P: G T A G C G G C G

T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G

Boyer-Moore: Preprocessing

Pre-calculate skips for all possible mismatch scenarios!
For bad character rule and *P* = GTAGCGGCG

P

Σ	G	T	A	G	C	G	G	C	G
A	0	1	-	0	1	2	3	4	5
C	0	1	2	3	-	0	1	-	0
G	-	0	1	-	0	-	-	0	-
T	0	-	0	1	2	3	4	5	6

P: G T A G C G G C G

T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G

Boyer-Moore: Preprocessing

Pre-calculate skips for all possible mismatch scenarios!
For bad character rule and *P* = GTAGCGGCG

P

Σ	G	T	A	G	C	G	G	C	G
A	0	1	-	0	1	2	3	4	5
C	0	1	2	3	-	0	1	-	0
G	-	0	1	-	0	-	-	0	-
T	0	-	0	1	2	3	4	5	6

P: G T A G C G G C G

T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G

Boyer-Moore: Preprocessing

Pre-calculate skips for all possible mismatch scenarios!
For bad character rule and P = GTAGCGGCG

P

Σ	G	T	A	G	C	G	G	C	G
A	0	1	-	0	1	2	3	4	5
C	0	1	2	3	-	0	1	-	0
G	-	0	1	-	0	-	-	0	-
T	0	-	0	1	2	3	4	5	6

P: G T A G C G G C G

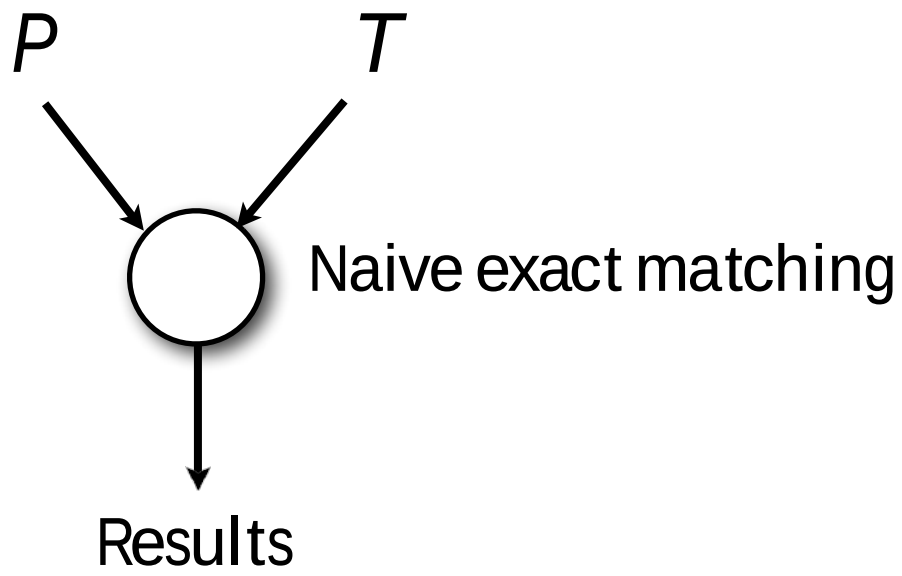
T: G T T A T A G C T G A T C G C G G C G T A G C G G C G A A
P: G T A G C G G C G

Boyer-Moore: Preprocessing

As with bad character rule, good suffix rule skips can be precalculated efficiently.

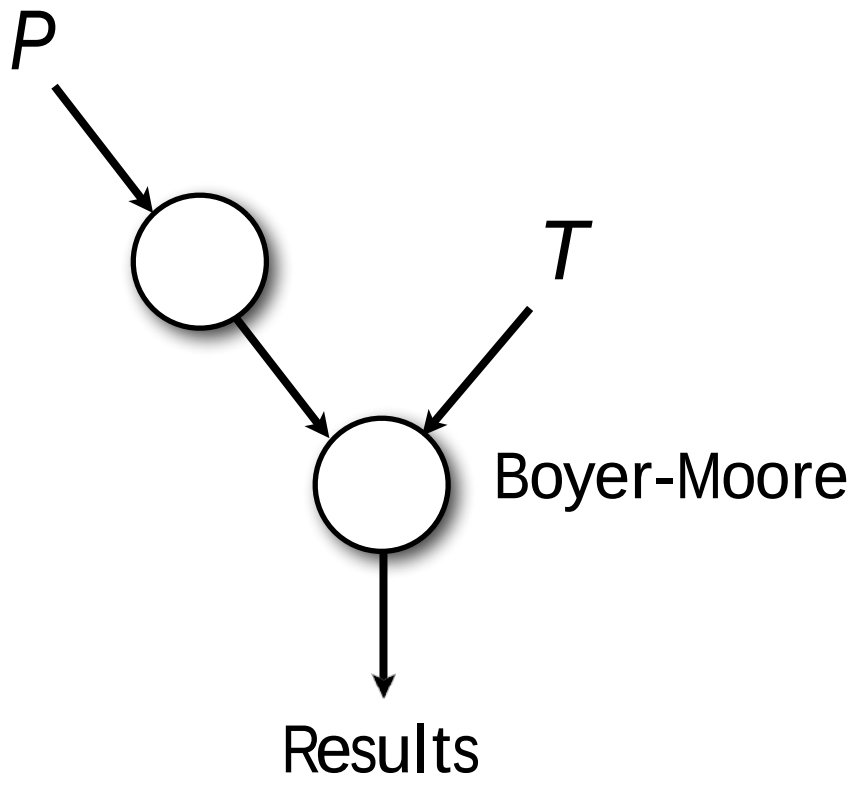
For both tables, the calculations only consider *P*.
No knowledge of *T* is required.

Preprocessing: Naive algorithm



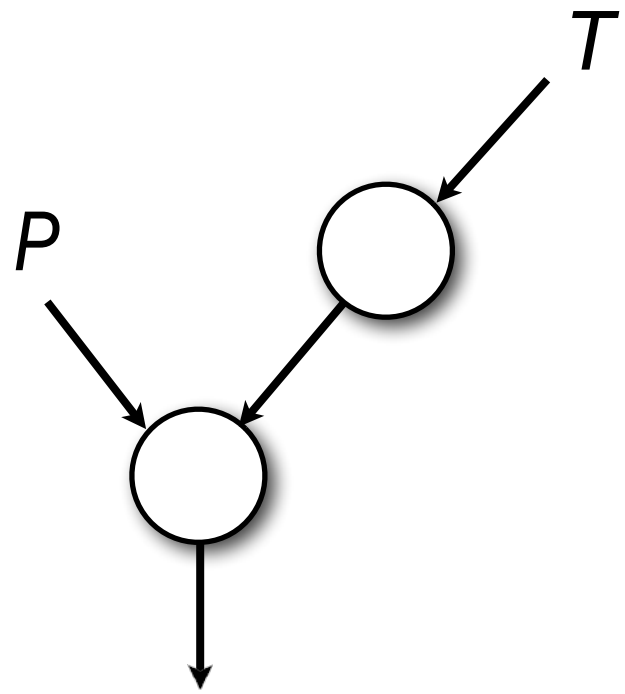
Preprocessing: Boyer-Moore

Preprocess *P*: *Make lookup tables for bad character & good suffix rules*



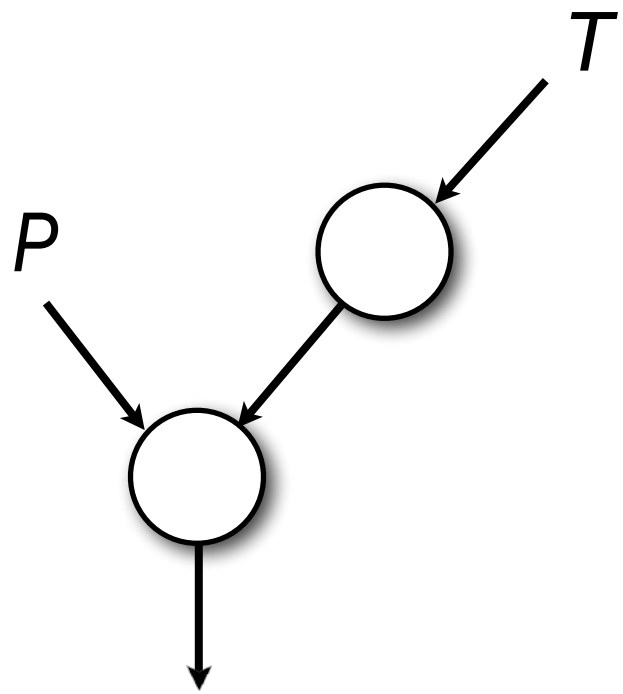
Preprocessing

Preprocess *T (offline):*
Genome or sequence that does not have frequent changes.



Preprocessing

Algorithm that preprocesses *T* is *offline*. Otherwise, algorithm is *online*.



Online or offline?

- Naïve algorithm
- Boyer-Moore
- Web search engine
- Read alignment

You have a pattern P and you expect to receive several texts T1, T2, T3, ..., and you would like to match P against each text as it arrives. It is better to use an algorithm that:

Preprocesses T **Preprocesses P** **Preprocesses neither P nor T**

An online algorithm does not:

Preprocess T **Preprocess P** **Preprocess at all**