Boyer-Moore

Ben Langmead

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Department of Computer Science

Can we improve on the naïve algorithm?

P: `word`

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

```
       word
```

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

P: `word`

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

```
       word
        word   skip!
         word   skip!
          word
```

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

P: **`word`**

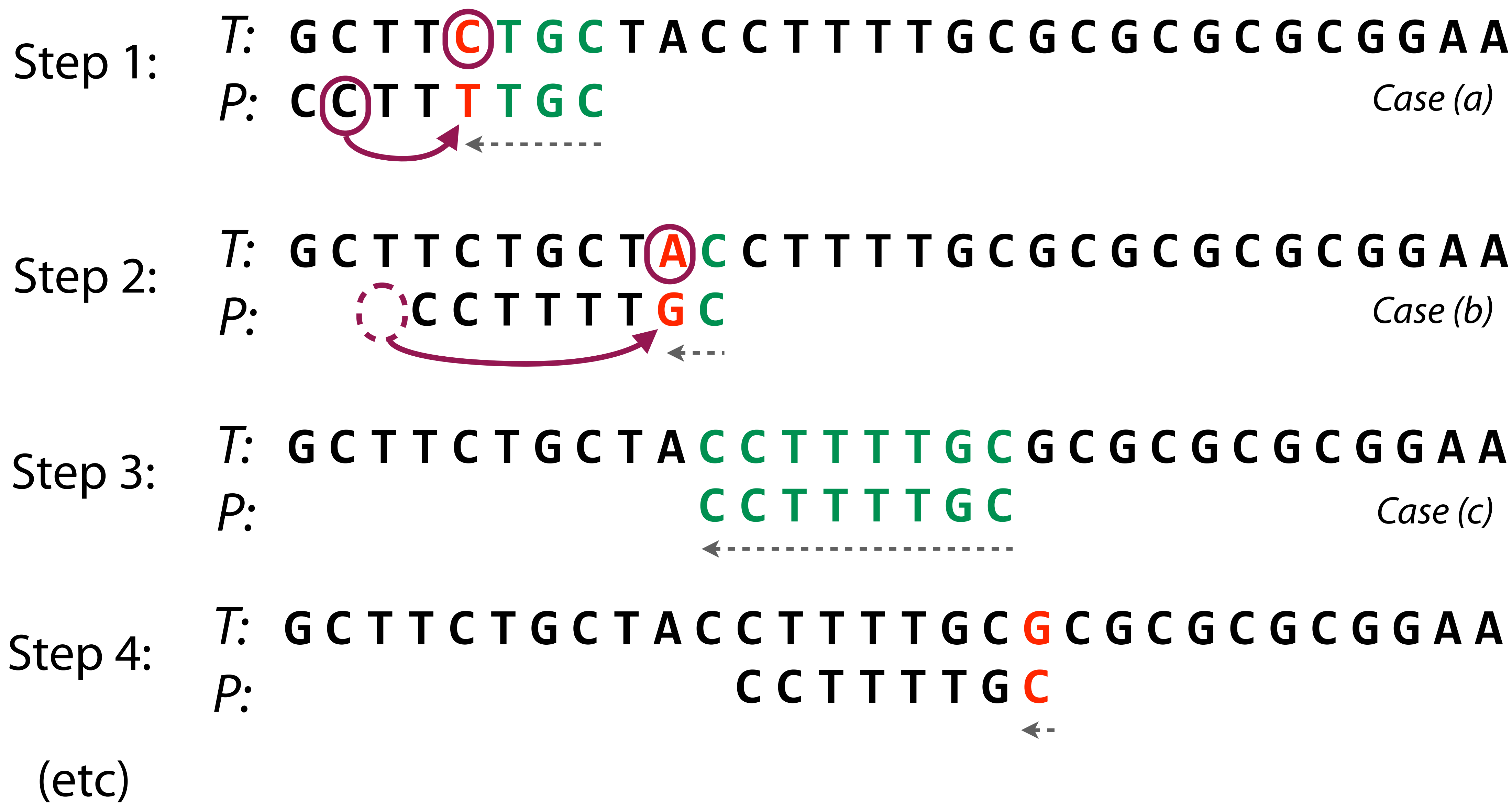
T: **`There would have been a time for such a word`**

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

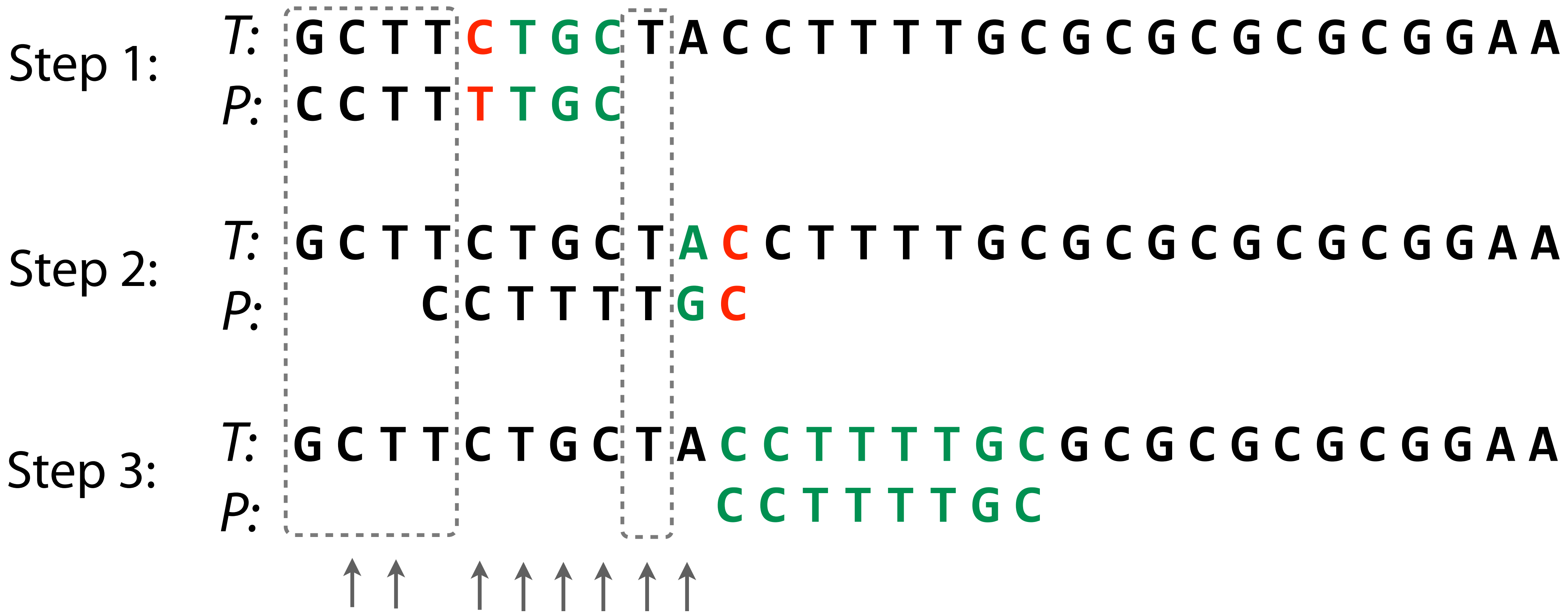
Boyer-Moore: Bad character rule

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



Boyer-Moore: Bad character rule



Up to step 3, we skipped 8 alignments

5 characters in *T* were never looked at

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*

Step 1:

```
                   t
*T*: C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A
*P*: C T T A C T T A C
```

Step 2:

```
                         t
*T*: C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A
*P*:           C T T A C T T A C
```

Step 3:

```
*T*: C G T G C C T A C T T A C T T A C T T A C T T A C G C G A A
*P*:                 C T T A C T T A C
```

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*

Step 1:

t

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

P: C T T A C T T A C

t occurs *in its entirety* to the left within *P*

Step 2:

t

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

P: C T T A C T T A C

prefix of *P* matches a *suffix* of *t*

Step 3:

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

P: C T T A C T T A C

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?

```
T: GTTATAGCTGATCGCGGCGTAGCGGCGAA
P:         GTAGCGGCG
```

bad char says skip 2, good suffix says skip 7

Take the maximum! (7)

Boyer-Moore: Putting it together

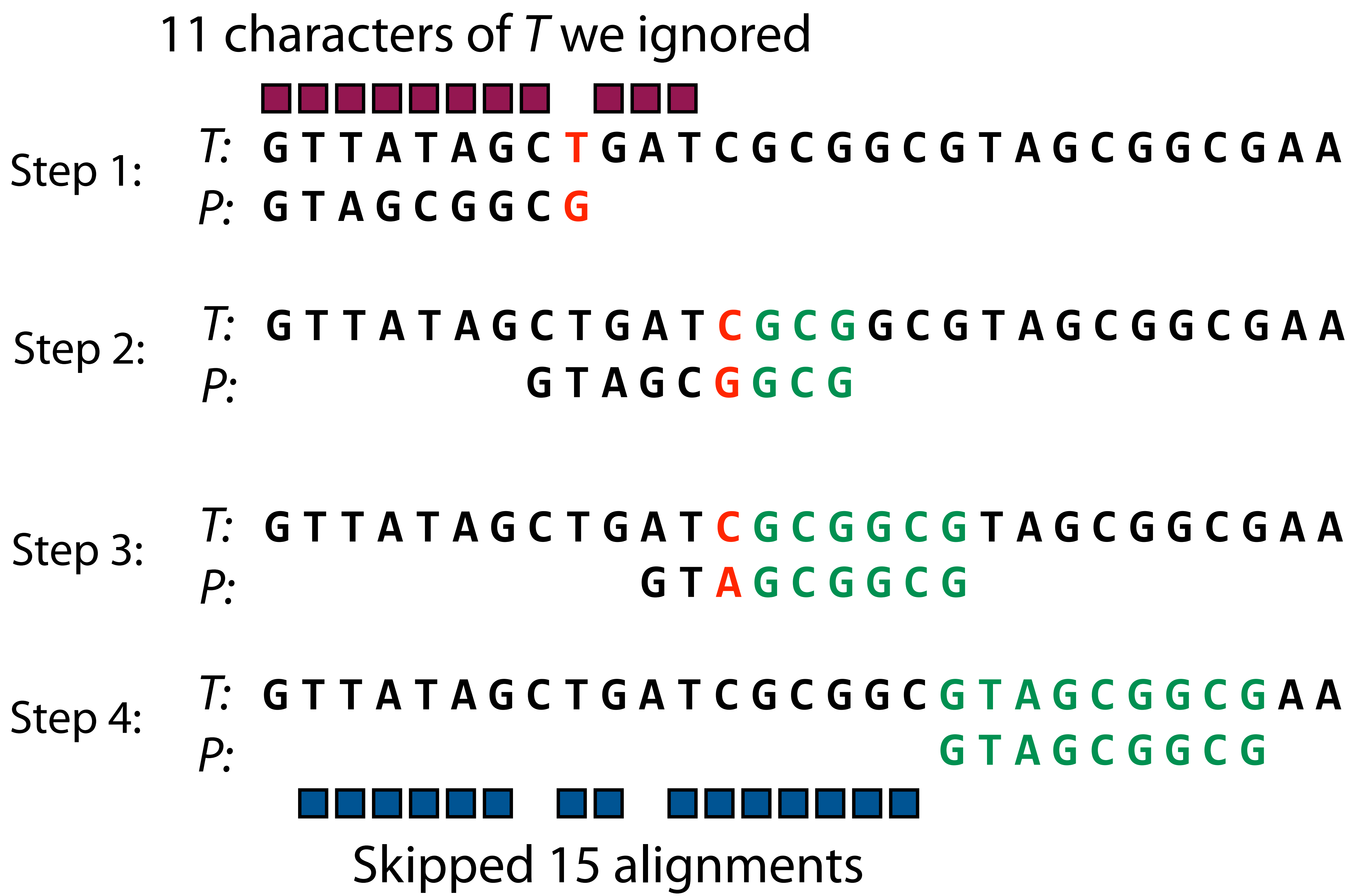
Use bad character or good suffix rule, *whichever skips more*

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 — bc: **6**, gs: 0 *bad character*

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 — bc: 0, gs: **2** *good suffix*

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 — bc: 2, gs: **7** *good suffix*

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


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

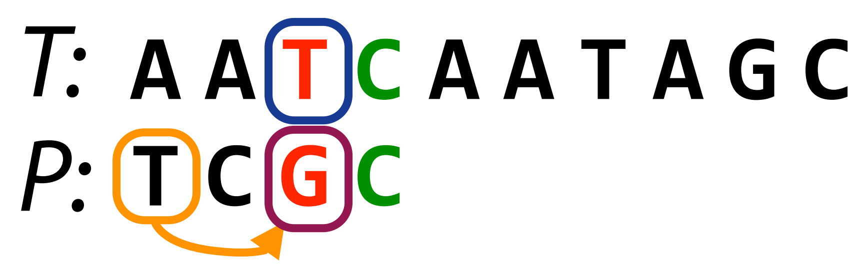
Pre-calculate skips for all possible mismatch scenarios!
For bad character rule and P = TCGC:

Σ \ P	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:

Σ \ *P*	T	C	G	C
A	0	1	2	3
C	0	-	0	-
G	0	1	-	0
T	-	0	1	2



This can be constructed efficiently. See Gusfield 2.2.2.

Boyer-Moore: Preprocessing

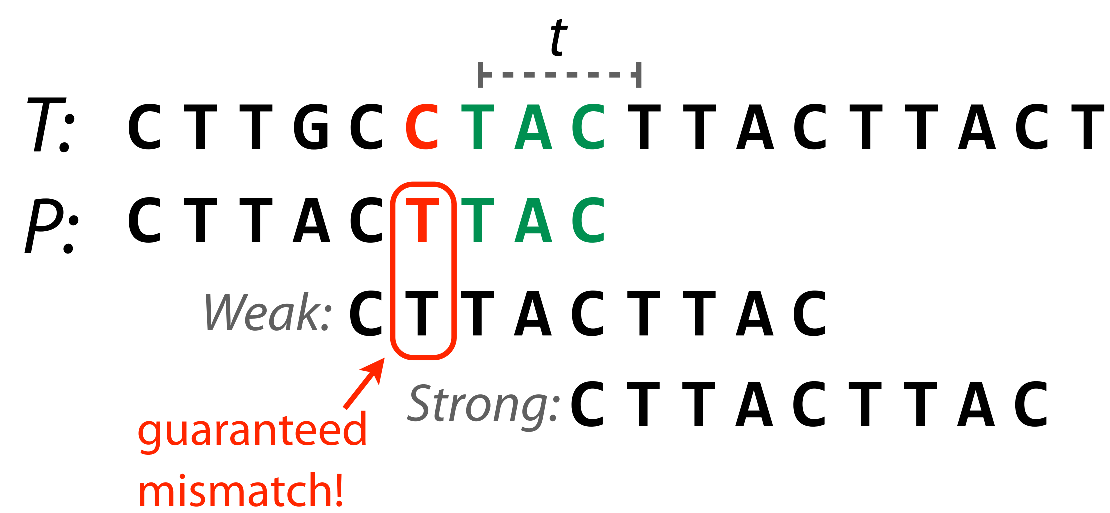
As with bad character rule, good suffix rule skips can be precalculated efficiently. See Gusfield 2.2.4 and 2.2.5.

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

We'll return to preprocessing soon!

Boyer-Moore: Good suffix rule

We learned the *weak* good suffix rule; there is also a *strong* good suffix rule



Strong good suffix rule skips more than weak, at no additional penalty

Strong rule is needed for proof of Boyer-Moore's O(n + m) worst-case time.
Gusfield discusses proof(s) in first several sections of ch. 3

Aside: Big-O notation

For review, see Jones & Pevzner 2.8

$$O(n^2)$$

"big oh of n squared"

Asymptotic upper bound on worst-case growth

Boyer-Moore: Worst case

Boyer-Moore, with refinements in Gusfield, is $O(n + m)$ time

Given $n < m$, can simplify to $O(m)$

Is this better than naïve?

For naïve, worst-case # char comparisons is $n(m - n + 1)$

Boyer-Moore: $O(m)$, naïve: $O(nm)$

Reminder: $|P| = n$ $|T| = m$

Boyer-Moore: Best case

What's the best case?

P: **bbbb**

T: **aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa**

bbbb **bbbb** **bbbb** **bbbb** **bbbb** **bbbb**

bbbb **bbbb** **bbbb** **bbbb** **bbbb**

Every alignment yields immediate mismatch and bad character rule skips *n* alignments

How many character comparisons? *floor(m / n)*

Naive vs Boyer-Moore

As m & n grow, # characters comparisons grows with...

$|P| = n \quad |T| = m$

	Naïve matching	Boyer-Moore
Worst case	m·n	m
Best case	m	m / n

Performance comparison

Simple Python implementations of naïve and Boyer-Moore:

	Naïve matching		Boyer-Moore		
	# character comparisons	wall clock time	# character comparisons	wall clock time	
P: "tomorrow" **T**: Shakespeare's complete works	5,906,125	2.90 s	785,855	1.54 s	17 matches $\mid T \mid = 5.59$ M
P: 50 nt string from Alu repeat* **T**: Human reference (hg19) chromosome 1	307,013,905	137 s	32,495,111	55 s	336 matches $\mid T \mid = 249$ M

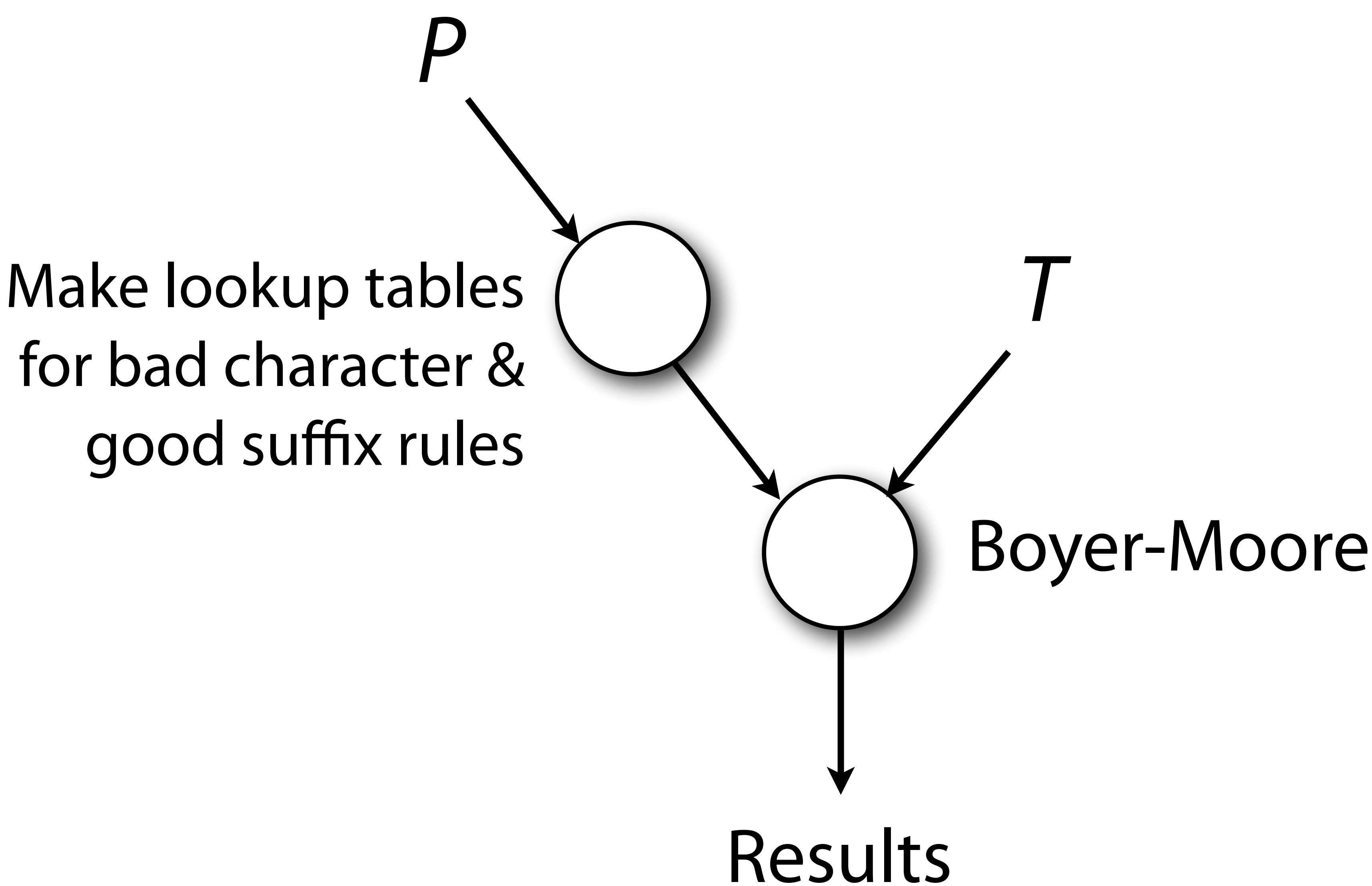
* GCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGG

Boyer-Moore implementation

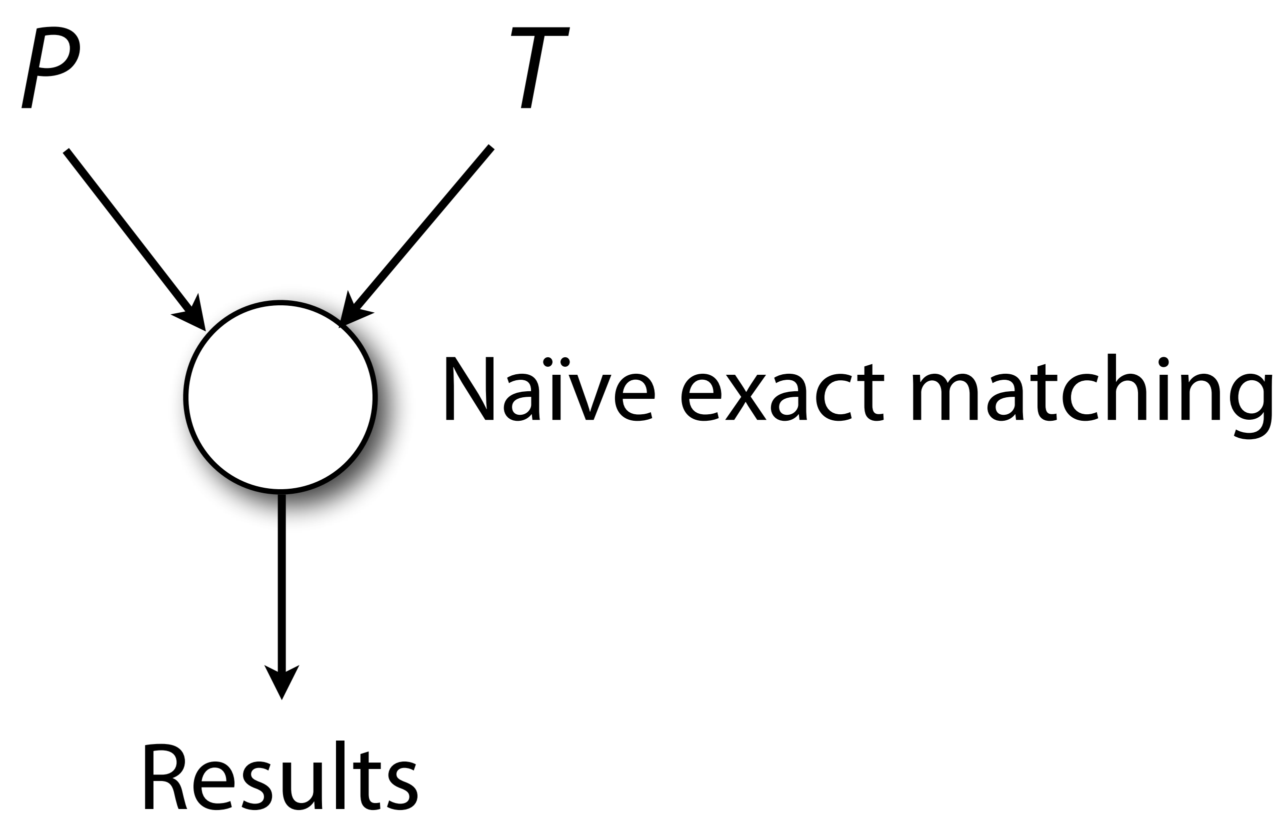
http://j.mp/CG_BoyerMoore

```
def boyer_moore(p, p_bm, t):
    """ Do Boyer-Moore matching """
    i = 0
    occurrences = []
    while i < len(t) - len(p) + 1:  # left to right
        shift = 1
        mismatched = False
        for j in range(len(p)-1, -1, -1):  # right to left
            if p[j] != t[i+j]:
                skip_bc = p_bm.bad_character_rule(j, t[i+j])
                skip_gs = p_bm.good_suffix_rule(j)
                shift = max(shift, skip_bc, skip_gs)
                mismatched = True
                break
        if not mismatched:
            occurrences.append(i)
            skip_gs = p_bm.match_skip()
            shift = max(shift, skip_gs)
        i += shift
    return occurrences
```

Preprocessing: Boyer-Moore



Preprocessing: Naïve algorithm



Preprocessing: Boyer-Moore

Preprocessing: trade one-time cost for reduced work overall via *reuse*

Boyer-Moore preprocesses *P* into lookup tables that are *reused*

reused for each alignment of *P* to T_1

If you later give me T_2, I *reuse* the tables to match *P* to T_2

If you later give me T_3, I *reuse* the tables to match *P* to T_3

...

Cost of preprocessing is *amortized* over alignments & texts