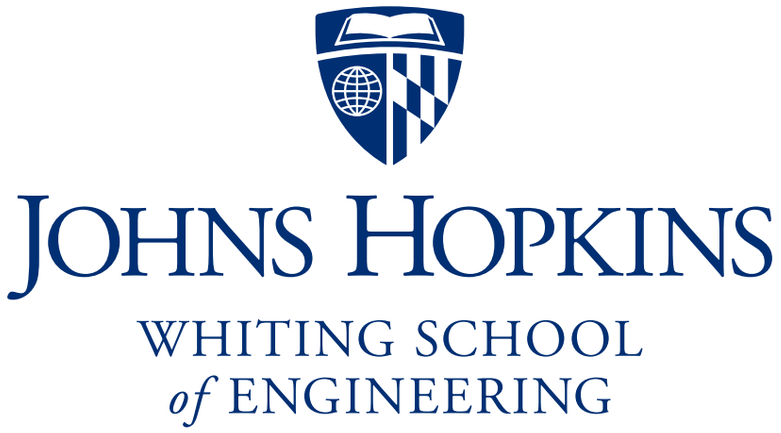
Strings, matching, Boyer-Moore

Ben Langmead



Resources

Gusfield, Dan. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, 1997.

iPython notebooks:
https://github.com/BenLangmead/comp-genomics-class

Including notebooks on strings, exact matching, and Z algorithm

Strings are a useful abstraction...

Lots of data is string-like: books, web pages, files on your hard drive, sensor data, medical records, chess games, ...

Algorithms for one kind of string are often applicable to others:

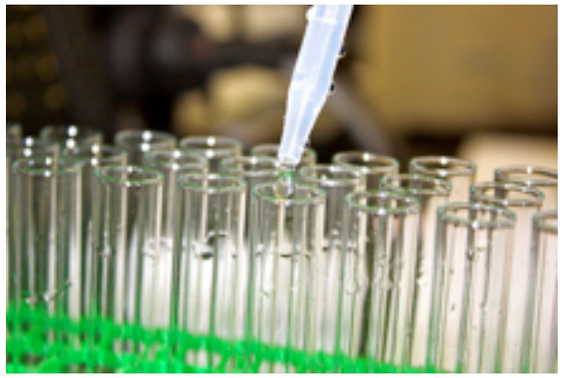
Regular expression matching is used to search files on your filesystem (grep), and to find "bad" network packets (snort)

Methods for indexing books and web pages (inverted indexing) can also be used to index DNA sequences

Methods for understanding speech (HMMs) can also be used to understand handwriting or identify genes in genomes

... but don't forget strings come from somewhere

Processes that give rise to real-world strings are complicated. It pays to understand them.



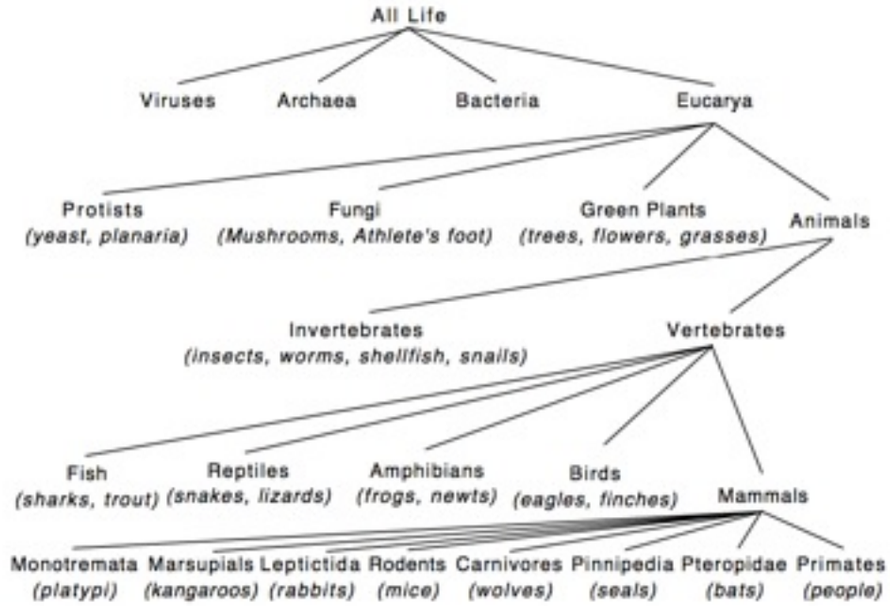
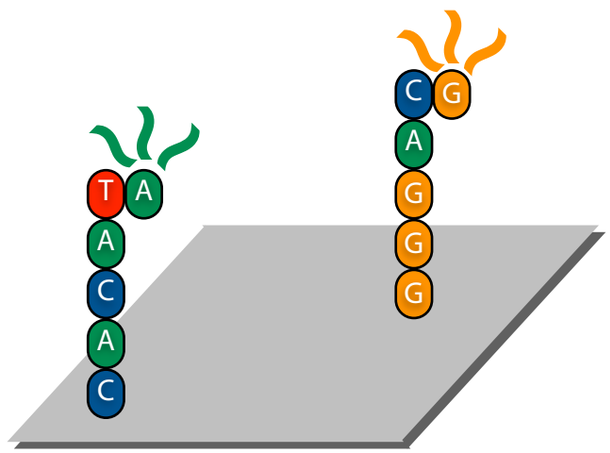


Figure from: Hunter, Lawrence. "Molecular biology for computer scientists." *Artificial intelligence and molecular biology* (1993): 1-46.

1. Evolution: Mutation
Recombination
(Retro)transposition

2. Lab procedures: PCR
Cell line passages



3. Sequencing: Fragmentation bias
Miscalled bases

... and don't forget strings have structure

One way to model a string-generating process is with coin flips:

{  = A,  = C,  = G,  = T }

But such strings lack internal patterns ("structure") exhibited by real strings

More than 40% of human genome is covered by *transposable elements*, which copy-and-paste themselves across the genome and mutate

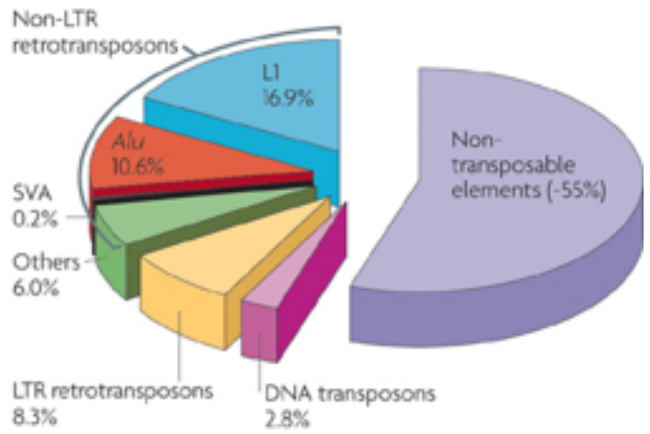
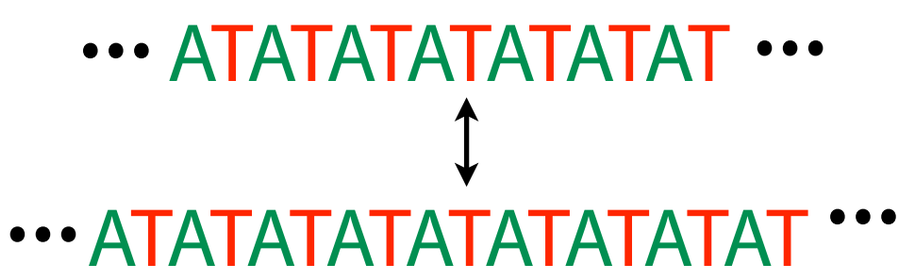


Image from: Cordaux R, Batzer MA. The impact of retrotransposons on human genome evolution. Nat Rev Genet. 2009 Oct;10(10):691-703

Slipped strand mispairing during DNA replication results in expansion or retraction of simple (*tandem*) repeats



String definitions

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

Characters are drawn from an alphabet Σ. We often assume Σ has $O(1)$ elements *.

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. $|\epsilon| = 0$

* but sometimes we'll consider $|\Sigma|$ explicitly

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

Python demo: http://nbviewer.ipython.org/6512698

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"

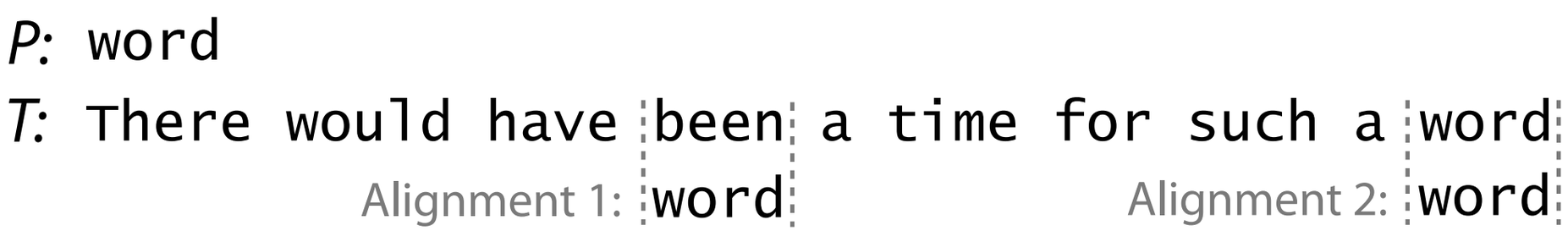
"cant" is a subsequence of "con**ca**te**n**a**t**e", but not a substring

Exact matching

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

Let $n = |P|$, and let $m = |T|$, and assume $n \leq m$

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



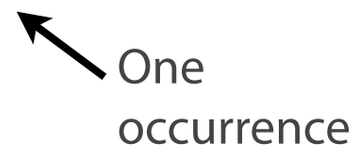
Exact matching

What's a simple algorithm for exact matching?

P: `word`

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

```
word word word word word word word word word word
 word word word word word word word word word word
  word word word word word word word word word word
   word word word word word word word word word word
```



Try all possible alignments. For each, check whether it's an occurrence. "Naïve algorithm."

Exact matching: naïve algorithm

```
def naive(p, t):
    occurrences = []
    for i in xrange(len(t) - len(p) + 1): # loop over alignments, L-to-R
        match = True
        for j in xrange(len(p)):          # loop over characters, L-to-R
            if t[i+j] != p[j]:            # character compare
                match = False             # mismatch; reject alignment
                break
        if match:
          occurrences.append(i)           # all chars matched; record
    return occurrences
```

Python demo: http://nbviewer.ipython.org/6513059

P: word

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

Exact matching: naïve algorithm

How many alignments are possible given *n* and *m* (| *P* | and | *T* |)?

m - *n* + 1

What is the greatest number of character comparisons possible?

n(*m* - *n* + 1)

the *least* possible?

m - *n* + 1

How many character comparisons in this example?

P: `word`

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

m - *n* mismatches, 6 matches

Exact matching: naïve algorithm

Greatest # character comparisons	Least:
$n(m - n + 1)$	$m - n + 1$

Worst-case time bound of naïve algorithm is $O(nm)$

In the best case, we do only ~ m character comparisons

Exact matching: slightly less naïve algorithm

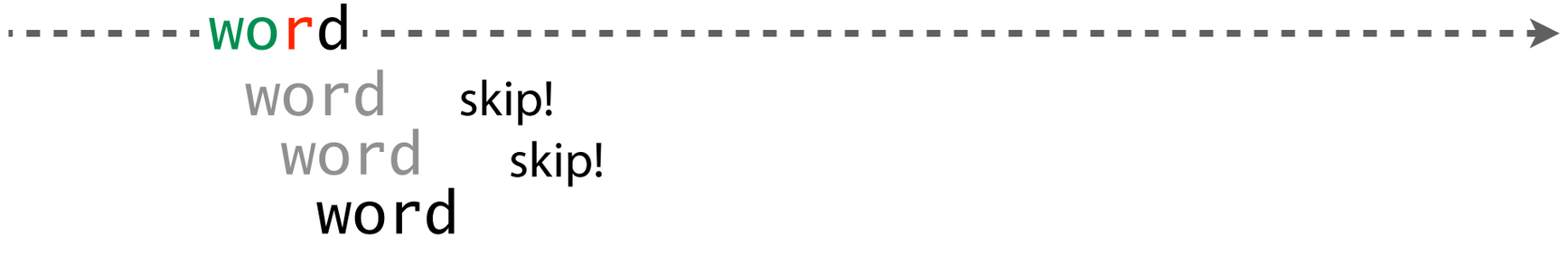
P: word
T: There would have been a time for such a word

We match w and o, then mismatch (r ≠ u)

Mismatched text character (u) doesn't occur in *P*

... since u doesn't occur in *P*, we can skip the next two alignments

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



Boyer-Moore

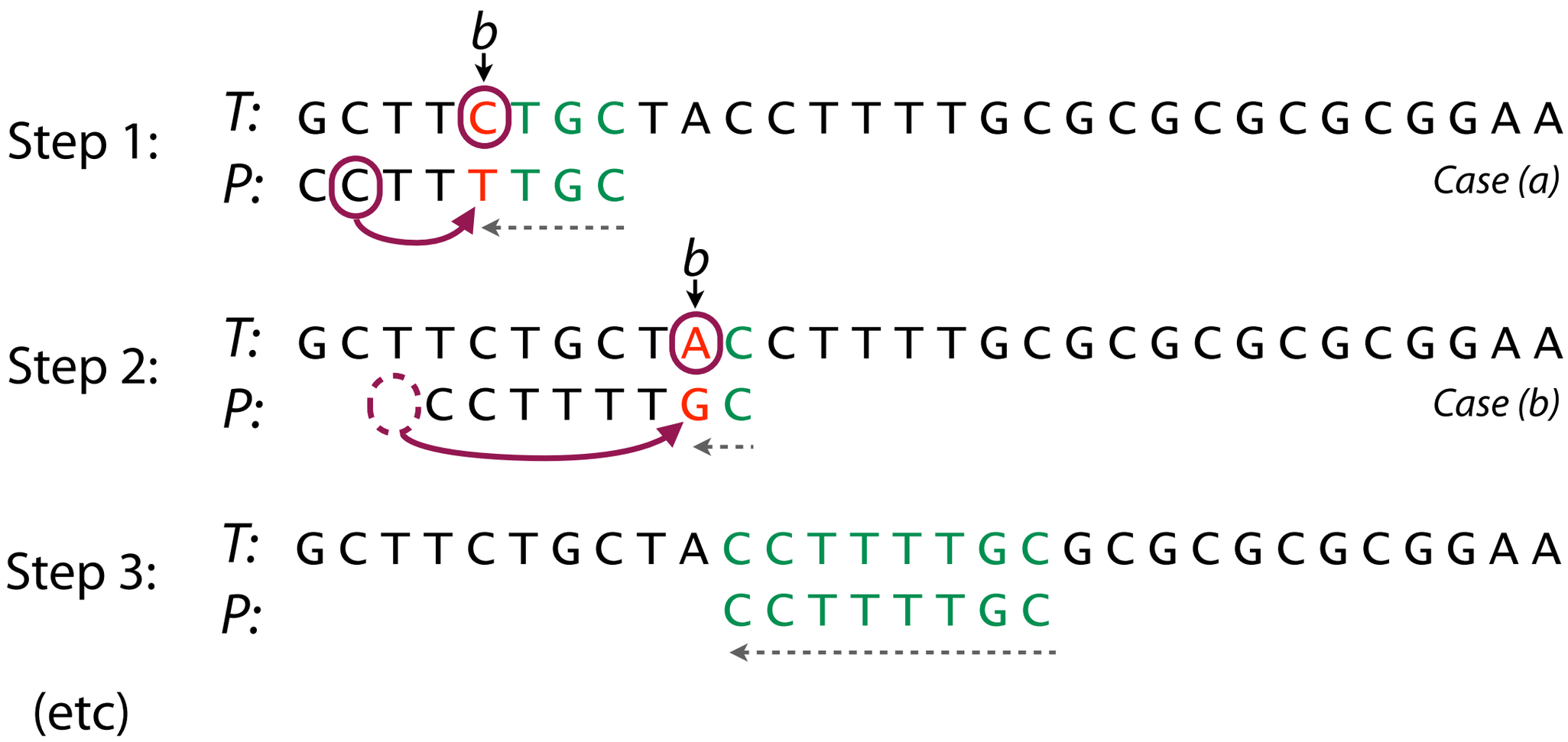
Use knowledge gained from character comparisons to skip future alignments that definitely won't match:

1. If we mismatch, use knowledge of the mismatched text character to skip alignments — "Bad character rule"
2. If we match some characters, use knowledge of the matched characters to skip alignments — "Good suffix rule"
3. Try alignments in one direction, then try 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.

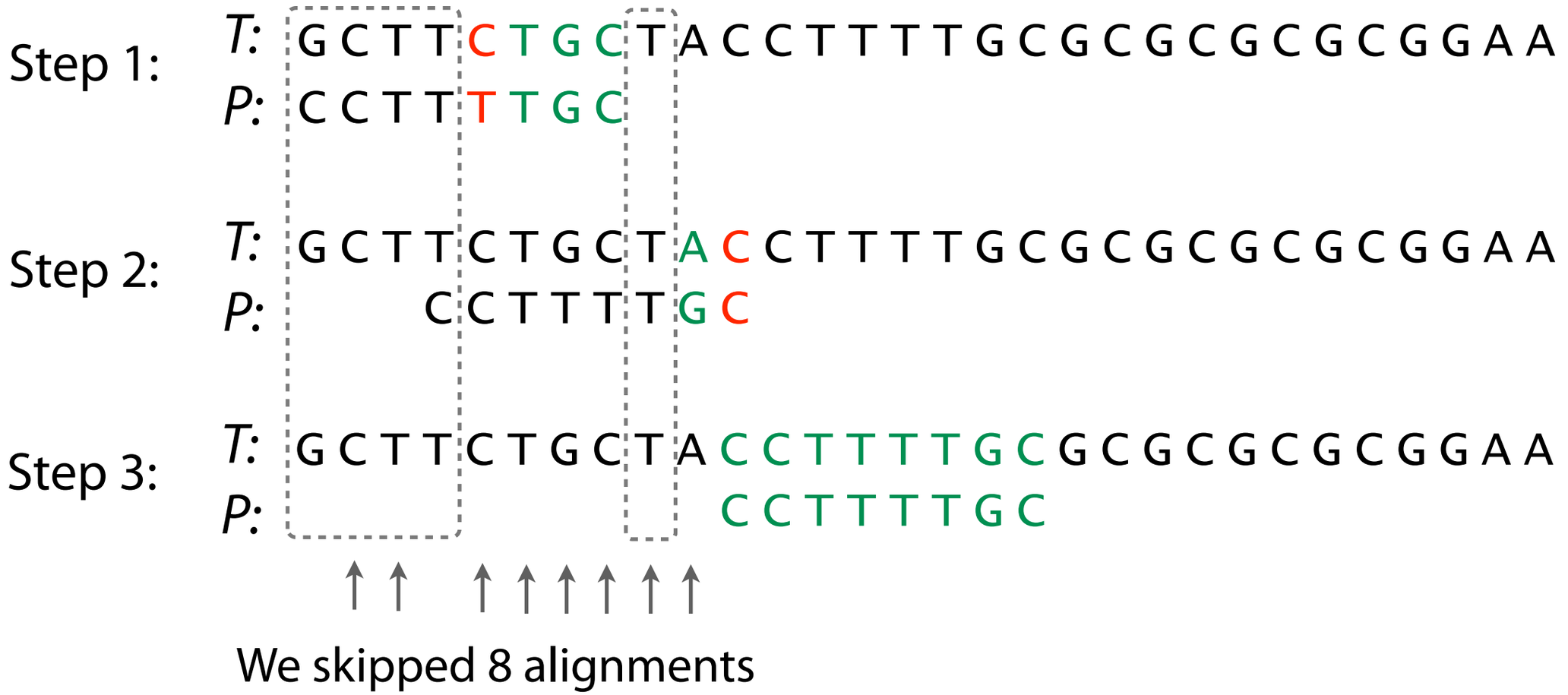
Boyer-Moore: Bad character rule

Upon mismatch, let *b* be the mismatched character in *T*. Skip alignments until (a) *b* matches its opposite in *P*, or (b) P moves past *b*.



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

Boyer-Moore: Bad character rule



In fact, there are 5 characters in *T* we never looked at

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

Boyer-Moore: Bad character rule preprocessing

As soon as *P* is known, build a $|\Sigma|$-by-*n* table. Say *b* is the character in *T* that mismatched and *i* is the mismatch's offset into *P*. The number of skips is given by element in *b*th row and *i*th column.

Gusfield 2.2.2 gives space-efficient alternative.

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

Boyer-Moore: Good suffix rule

Let *t* be the substring of *T* that matched a suffix of *P*. Skip alignments until (a) *t* matches opposite characters in *P*, or (b) a prefix of *P* matches a suffix of *t*, or (c) *P* moves past *t*, whichever happens first

Step 1:

Step 2:

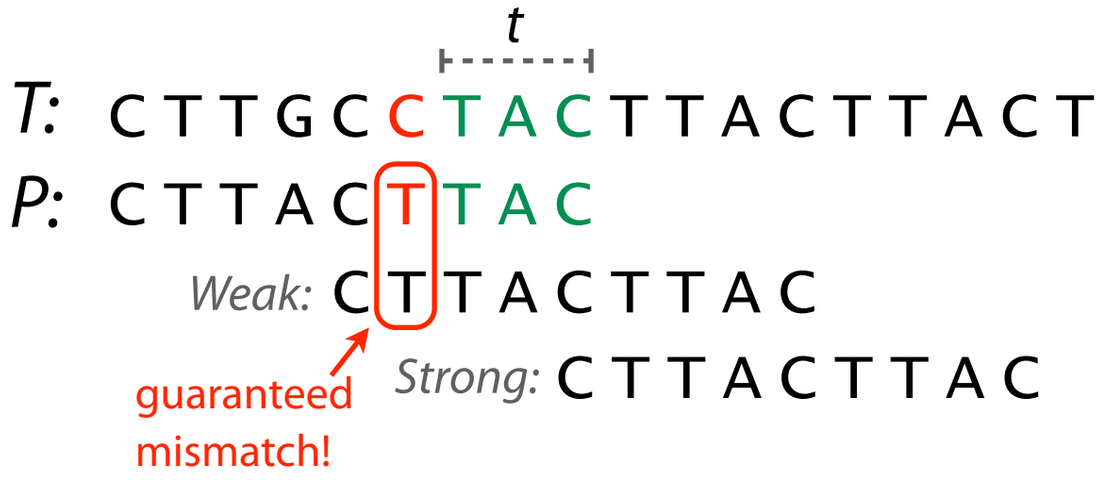
Step 3:

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

Boyer-Moore: Good suffix rule

Like with the bad character rule, the number of skips possible using the good suffix rule can be precalculated into a few tables (Gusfield 2.2.4 and 2.2.5)

Rule on previous slide is the *weak* good suffix rule; there is also a *strong* good suffix rule (Gusfield 2.2.3)



With the strong good suffix rule (and other minor modifications), Boyer-Moore is *O*(*m*) *worst-case* time. Gusfield discusses proof.

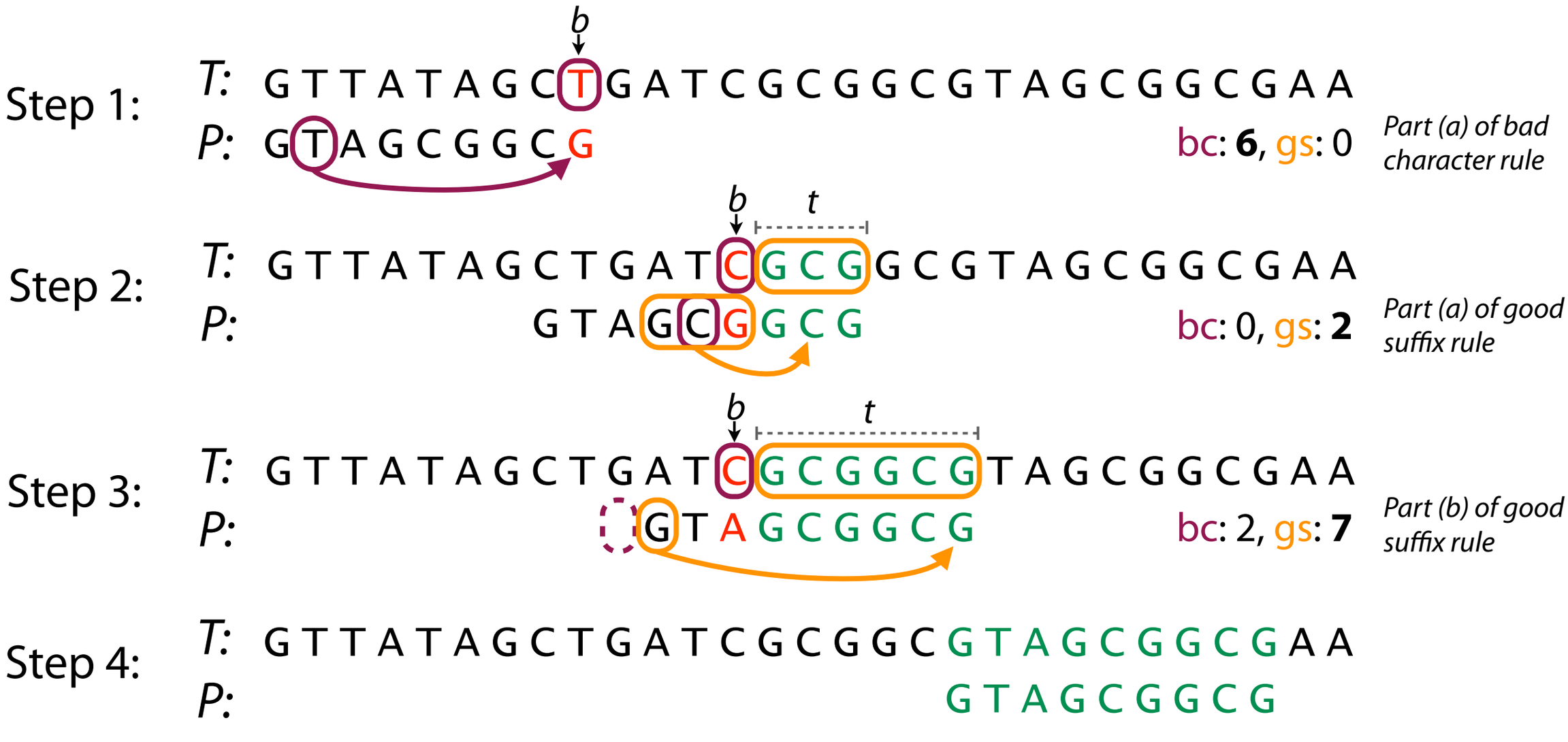
Boyer, RS and Moore, JS. "A fast string searching algorithm." *Communications of the ACM* 20.10 (1977): 762-772.

Boyer-Moore: Putting it together

After each alignment, use bad character or good suffix rule, whichever skips more

Bad character rule:
Upon mismatch, let *b* be the mismatched character in *T*. Skip alignments until (a) *b* matches its opposite in *P*, or (b) *P* moves past *b*.

Good suffix rule:
Let t be the substring of *T* that matched a suffix of *P*. Skip alignments until (a) *t* matches opposite characters in *P*, or (b) a prefix of *P* matches a suffix of *t*, or (c) *P* moves past *t*, whichever happens first.



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

Boyer-Moore: Putting it together

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

Up to now: 15 alignments skipped, 11 text characters never examined

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

Boyer-Moore: Worst and best cases

Boyer-Moore (or a slight variant) is *O*(*m*) worst-case time

What's the best case?

> Every character comparison is a mismatch, and bad character rule always slides *P* fully past the mismatch
>
> How many character comparisons? *floor(m / n)*

Contrast with naive algorithm

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

Performance comparison

Comparing simple Python implementations of naïve exact matching and Boyer-Moore exact matching:

	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 $\|T\| = 5.59$ 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 $\|T\| = 249$ M

* GCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGG