Sequence, String, and Tree

BCB 5200 Introduction Bioinformatics I

Fall 2017

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Saint Louis University

SAINT LOUIS UNIVERSITY™

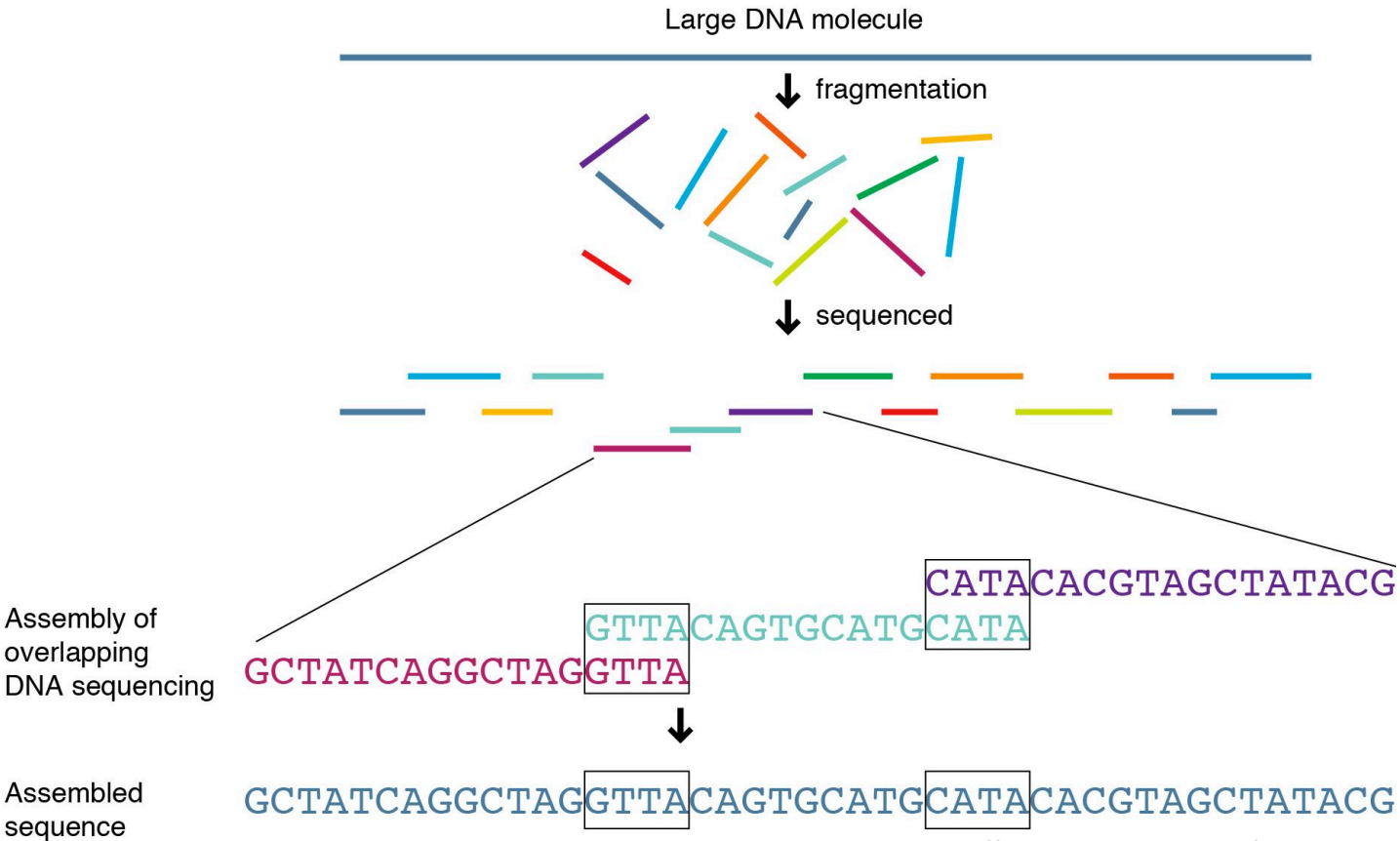
EST. 1818

Sequence Alignment

- In bioinformatics, a sequence alignment is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences.
- You learned about
  - Pairwise sequence alignment & Multiple sequence alignment
  - Local sequence alignments & Global sequence alignment
- Using
  - Dynamic Programming
  - Dot matrix
  - Hash
- For evolutionary relationship and more…

Sequencing a Genome

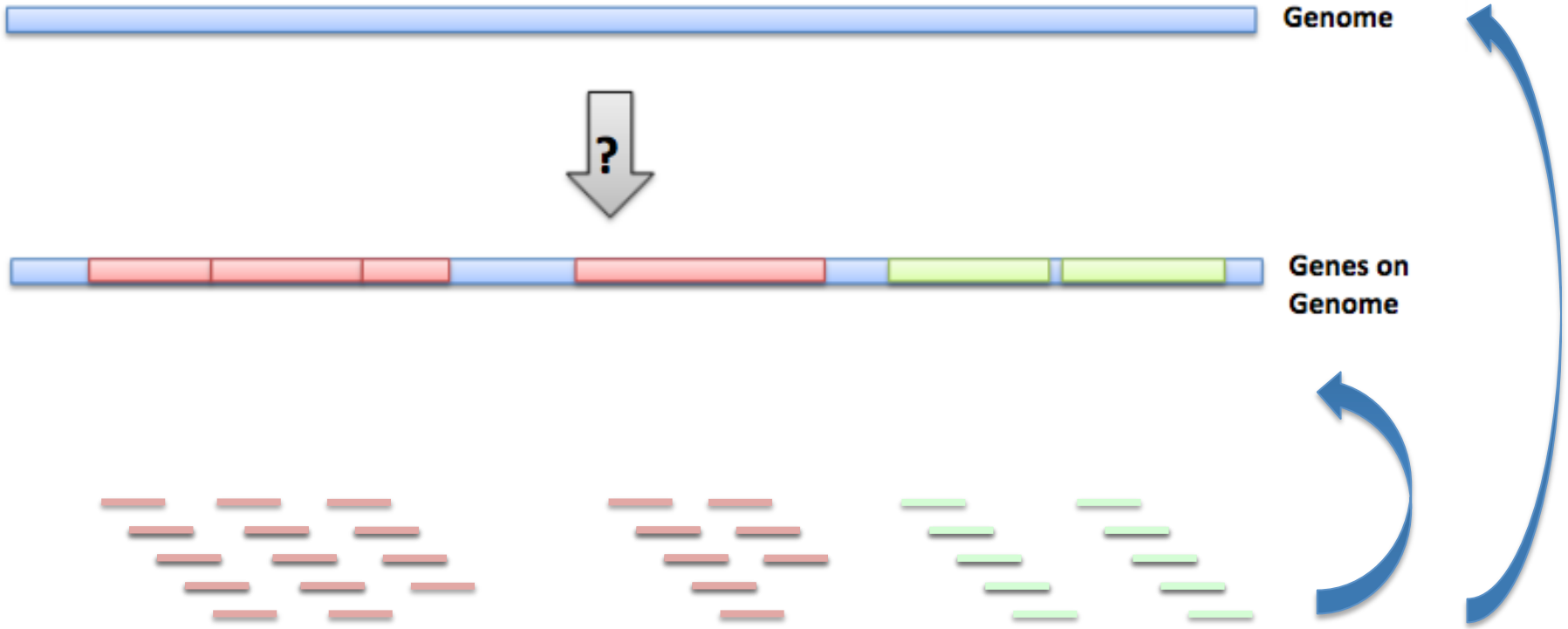
Most genomes are enormous (e.g., 3 billion base pairs in case of human). Current sequencing technology, on the other hand, only allows to generate ~60K base pairs at a time. This leads to some very interesting problems in bioinformatics...



http://knowgenetics.org/whole-genome-sequencing/

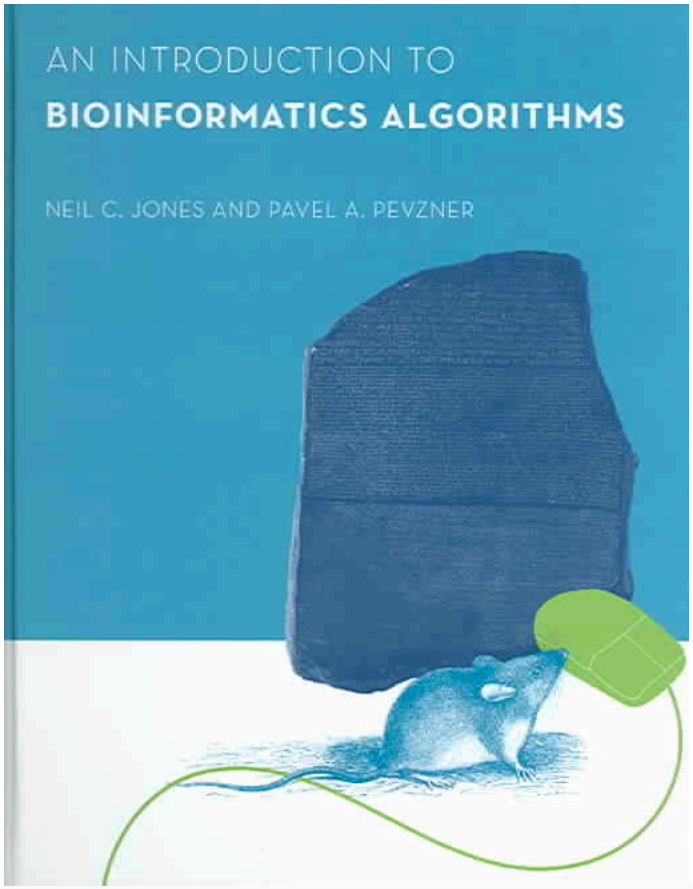
Finding Genes from Genome

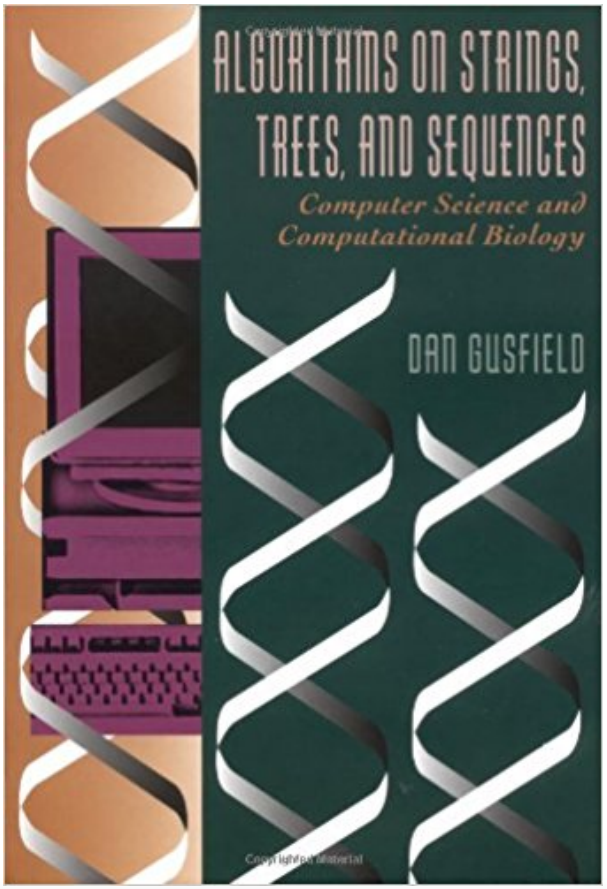
- To find out coding regions on genome sequences, we need to align the reads to the genes or genomes.



Sequence…String…Tree…Algorithms

- Most of bioinformatics application require sequence alignment, string match, tree and so on.
- You will learn bioinformatics algorithms in BCB5300 (Algorithms in Computational Biology)





Many CS Algorithms for Sequence Matching

Q: Where is GATTACA in the human genome?

CS Techniques:

- Brute Force
- Suffix Array, Suffix Tree
- Dynamic Programming
- *K*-mers
- Hash Table, MinHash
- Indexing, Burrows-Wheeler Transformation
- …..

String Matching

- A string S is a finite ordered list of characters
- String Matching:
  - Given text T & pattern P
  - Strings over 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}
  - Find some/all occurrences of P in T as substring
  - Static Data Structures: Preprocess T
  - Query: P
  - Goal: O(P) query, O(T) space

Substring, Prefix, Suffix

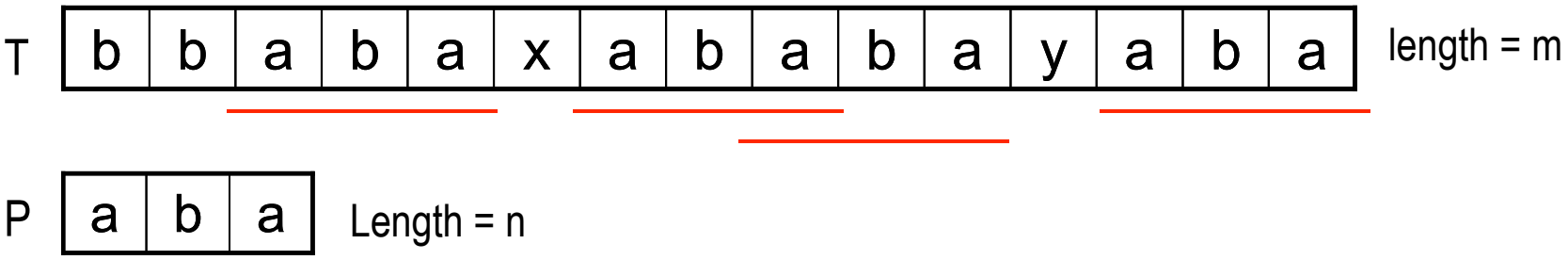
- S is substring of T if there exist (possibly empty) string u and v such as T=uSv
- S is prefix of T if there exist a string u such that T=Su.
- S is suffix of T if there exist a string u such that T=uS.

Python example

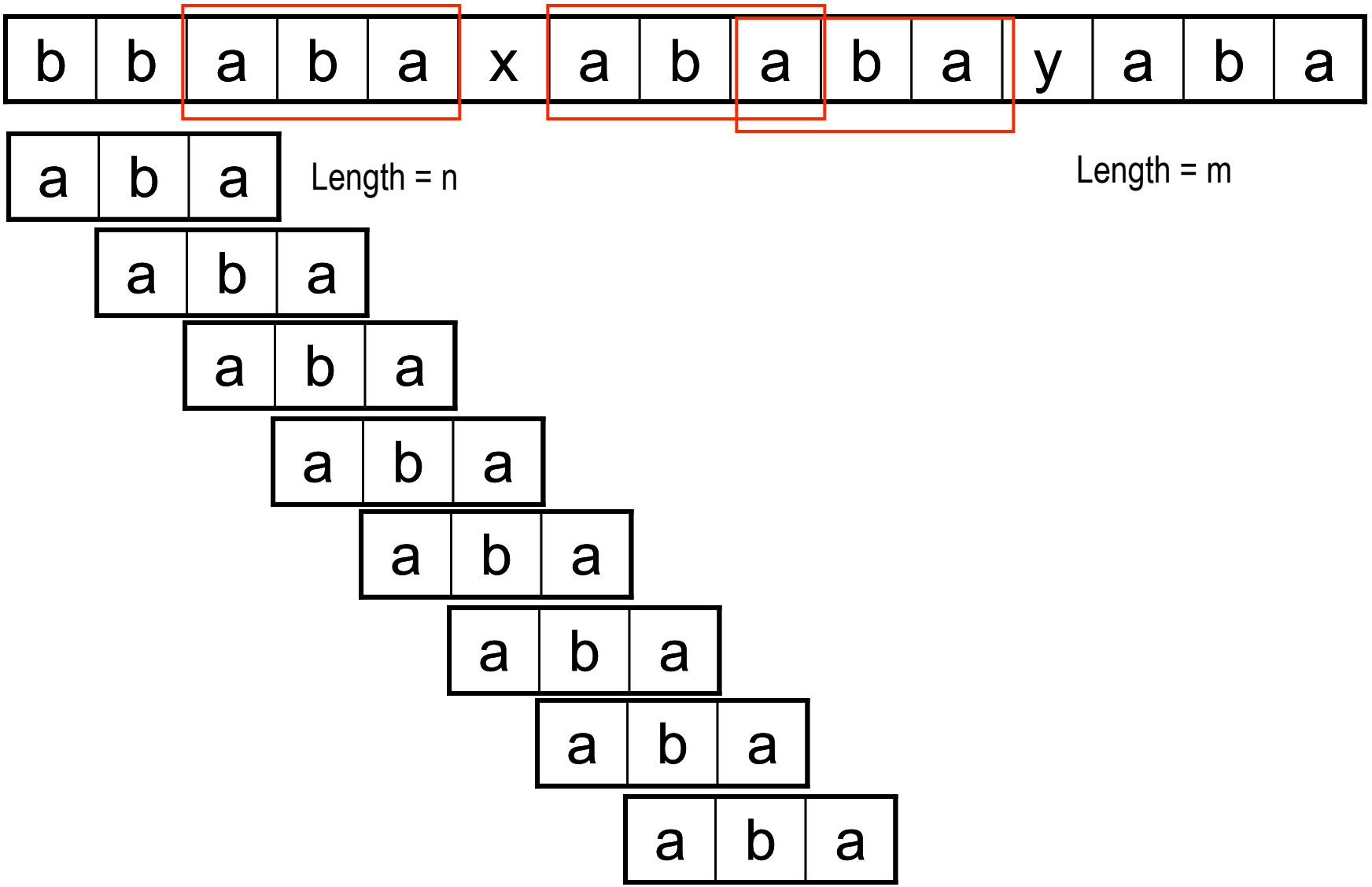
```
>>> st = 'ACGT'
>>> len(st) # getting the length of a string
>>> import random
>>> random.choice('ACGT') # generating a random nucleotide
>>> # now I'll make a random nucleotide string by concatenating random
nucleotides
>>> st = ''.join([random.choice('ACGT') for _ in xrange(40)])
>>> st
>>> st[1:3] # substring, starting at position 1 and extending up to but not
including position 3
# note that the first position is numbered 0
>>> st[0:3] # prefix of length 3
>>> st[:3] # another way of getting the prefix of length 3
>>> st[len(st)-3:len(st)] # suffix of length 3
>>> st[-3:] # another way of getting the suffix of length 3
>>> st1, st2 = 'CAT', 'ATAC'
>>> st1 + st2 # concatenation of 2 strings
```

Exact matching

- Text: a longer string T
- Pattern: a shorter string P
- Exact matching: find all occurrence of P in T



Exact matching: naïve algorithm



Exact matching: naïve algorithm

```
>>> t = 'Intro to Bioinformatics course is fun' # "text" - thing we search in
>>> p = 'fun' # "pattern" - thing we search for
>>>
>>> def naive(p, t):
...     assert len(p) <= len(t) # assume text at least as long as pattern
...     occurrences = []
...     # For each alignment of p to t
...     for i in xrange(0, len(t)-len(p)+1):
...         match = True # assume the pattern matches until proven wrong
...         # For each position of p
...         for j in xrange(0, len(p)):
...             if t[i+j] != p[j]:
...                 match = False # at least 1 char mismatches, so no match
...                 break
...         if match:
...             occurrences.append(i)
...     return occurrences
...
>>> naive(p, t)
[34]
>>> t[34:34+len(p)]
'fun'
>>> naive('needle', 'needleneedleneedle')
[0, 6, 12]
```

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?

$m - n + 1$

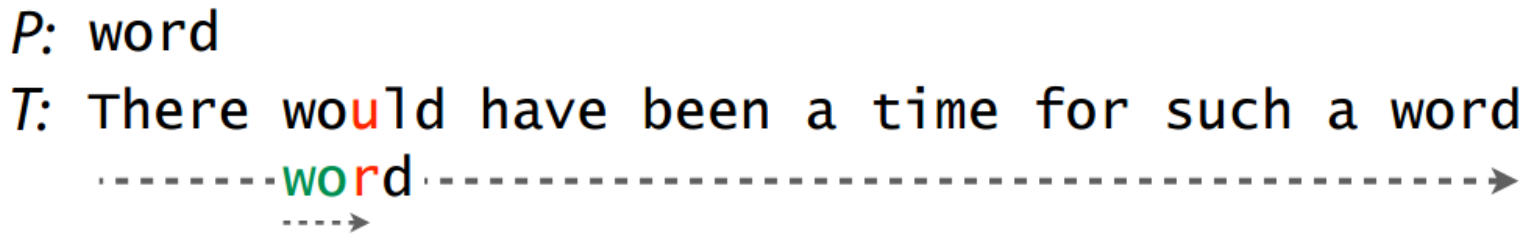
Exact matching: naïve algorithm

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

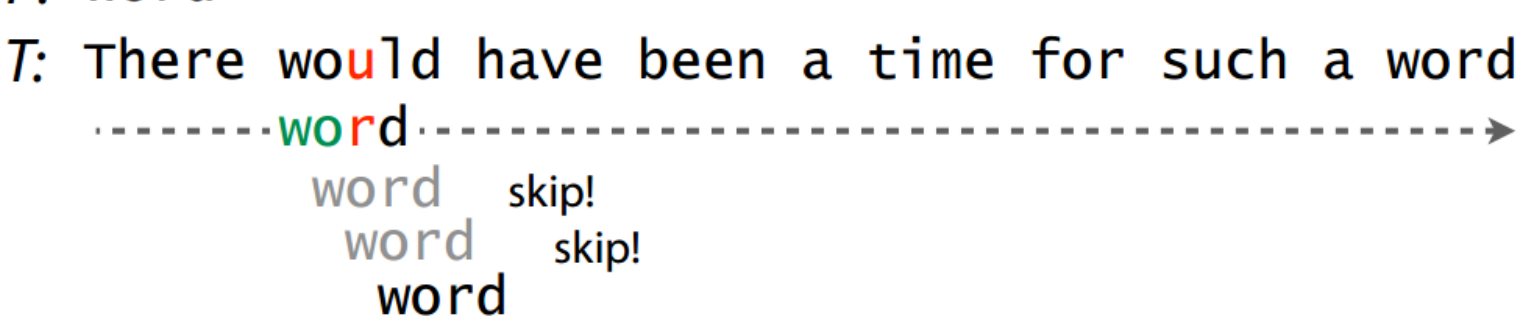
Exact matching: slightly less naïve algorithm



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



Boyer-Moore

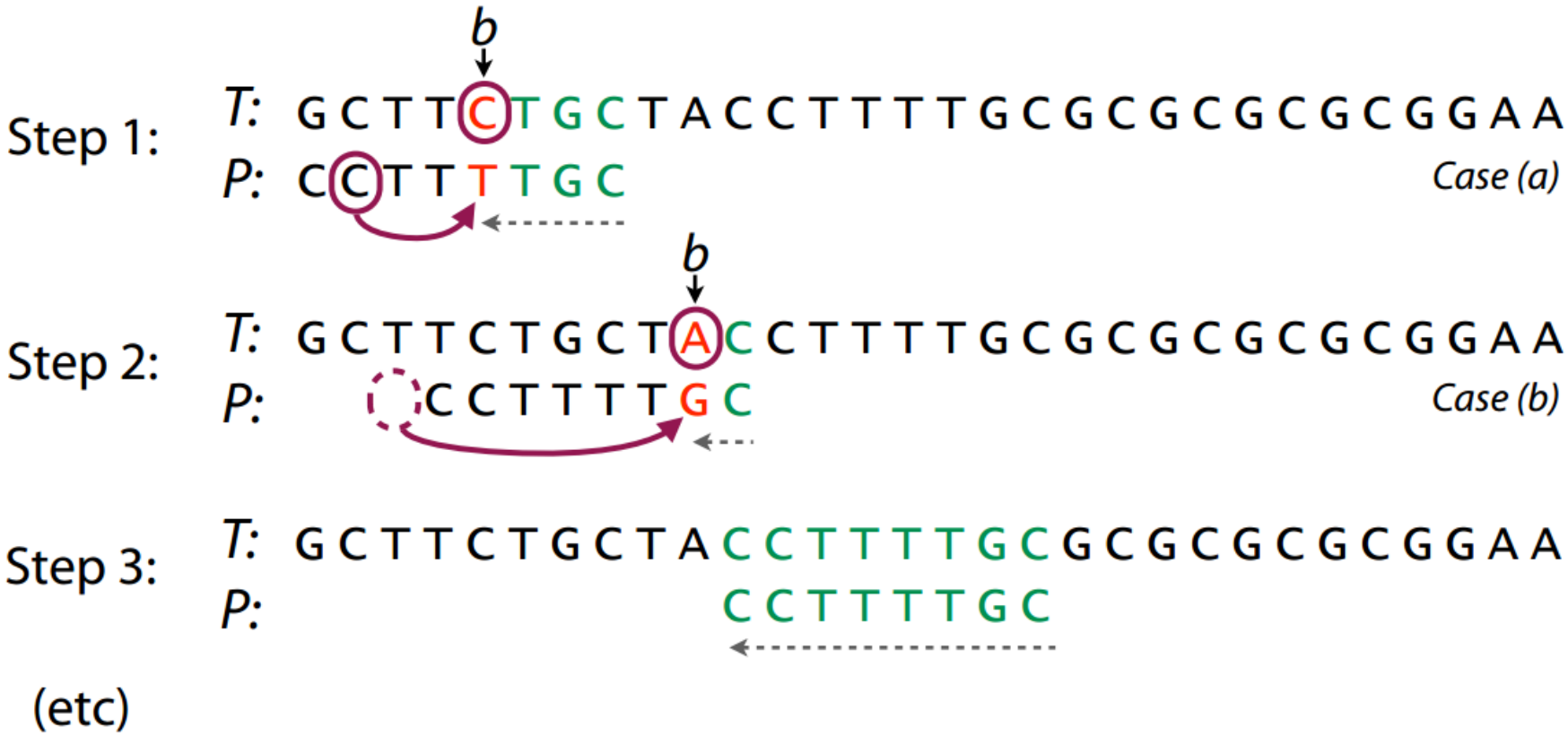
Use knowledge gained from character comparisons to skip future alignments that definitely won't match:

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

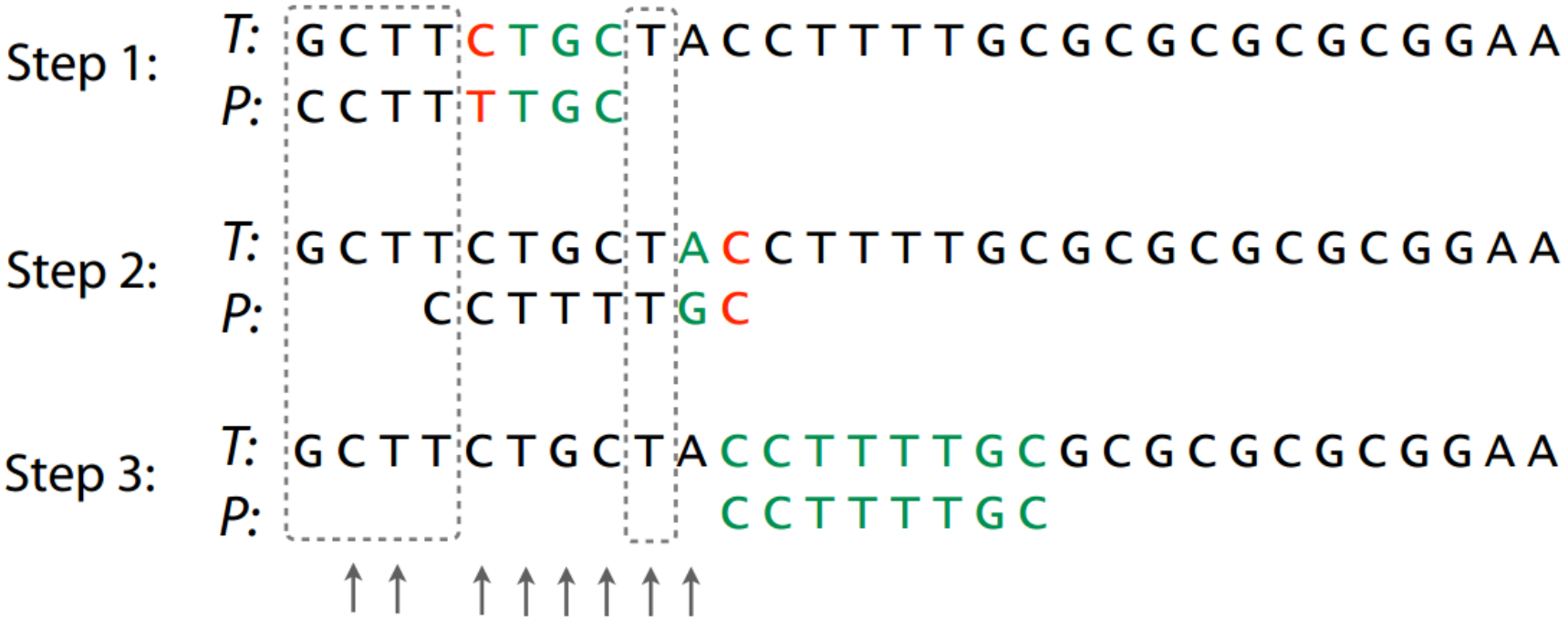
Boyer, RS and Moore, JS. "A fast string searching algorithm." *Communications of the ACM* 20.10 (1977): 762-772.

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*.



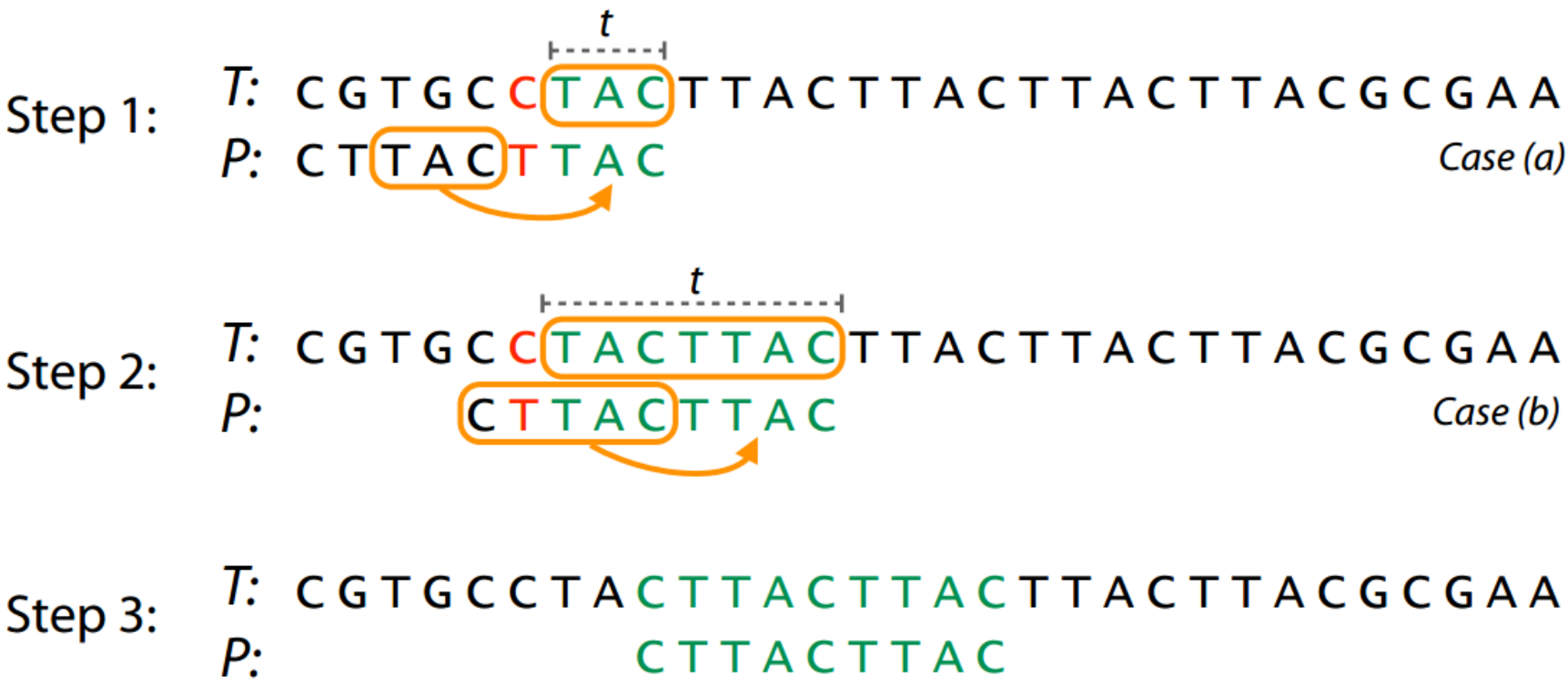
Boyer-Moore: Bad character rule



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

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

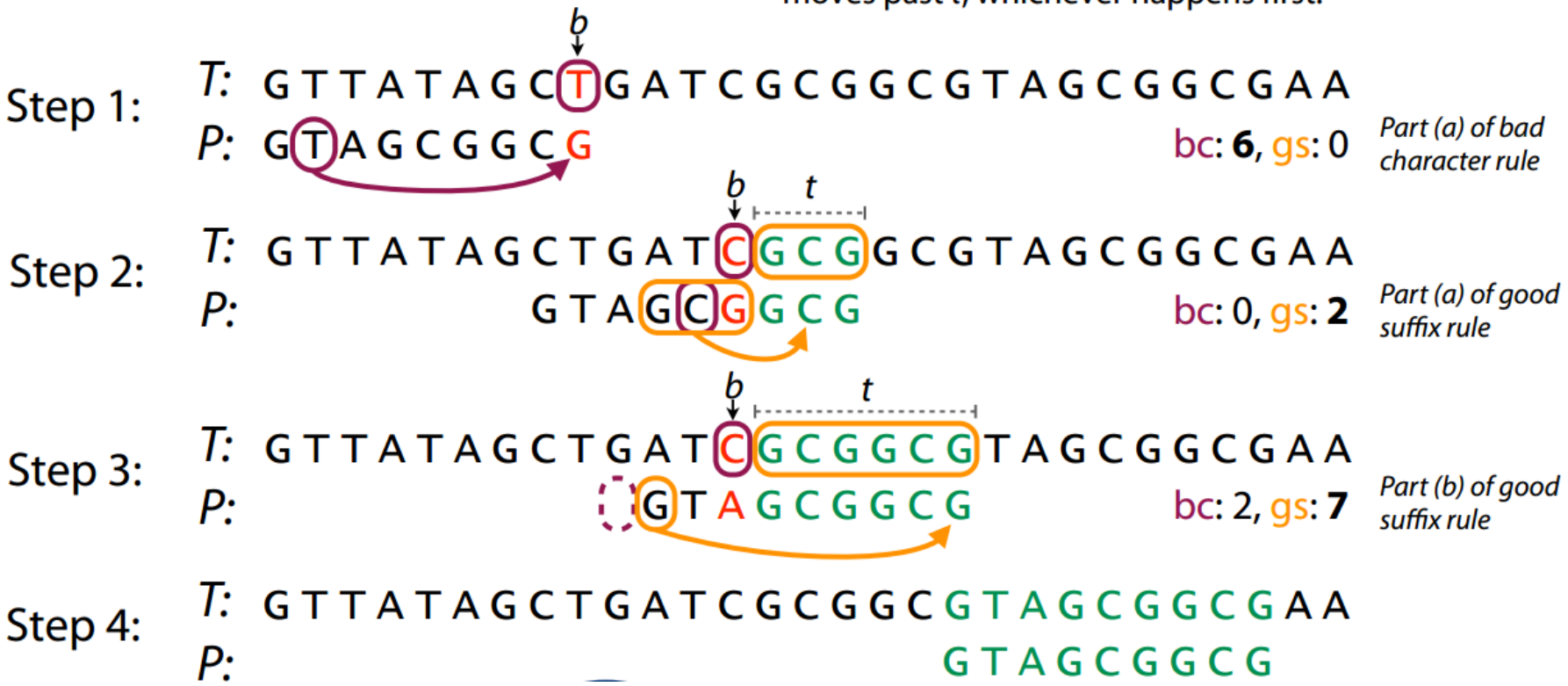


Boyer-Moore: Putting it together

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-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-Moore: Preprocessing

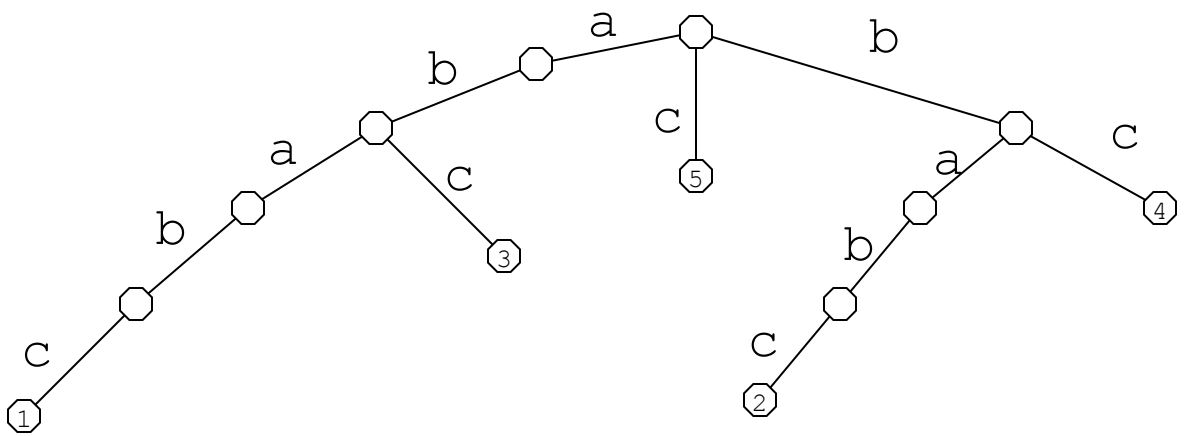
Pre-calculate skips. For bad character rule, $P = \text{TCGC}$:

P

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

Suffix Trie

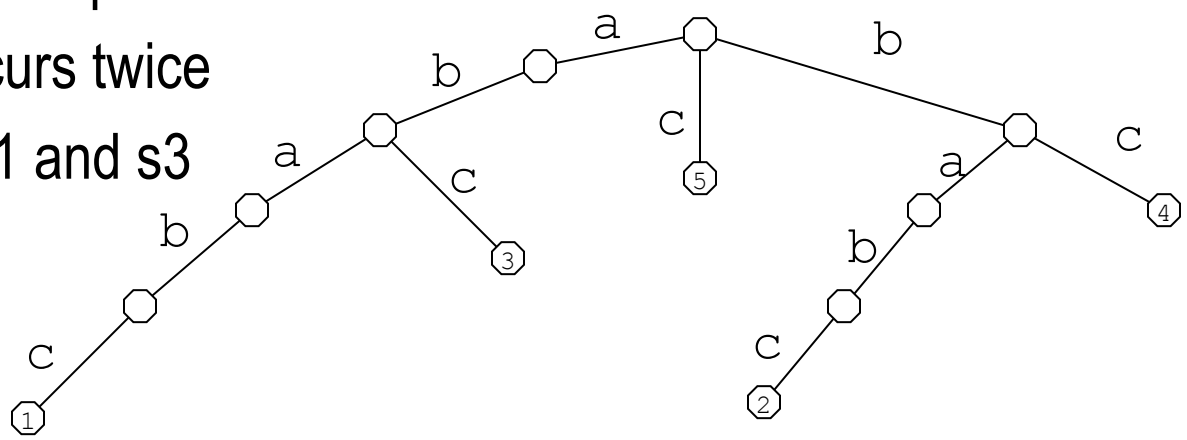
- Multi-way tree
  - Each branch is labeled with char
- If the trie is ready, match takes O(|pattern|) time
- Example: text S is ababc
  - s_1 = ababc
  - s_2 = babc
  - s_3 = abc
  - s_4 = bc
  - s_5 = c



Suffix Trie

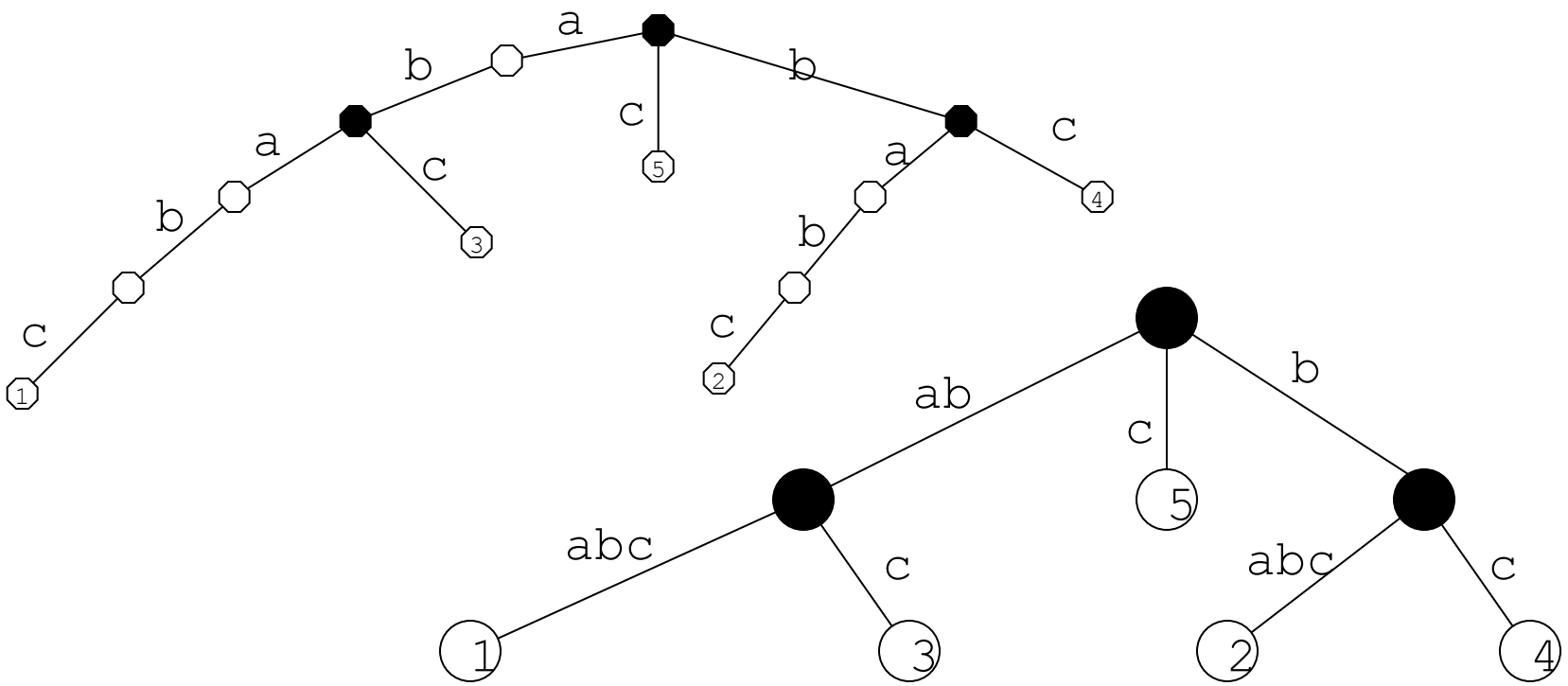
- Suffix trie takes $O(|S|^2)$ space
- Each step of search for match takes constant time
  - If no branch matches char, we fail
- Leaf holds name of suffix
- We may have multiple matches
  - String ab occurs twice
    - Prefix of s1 and s3

s_1 = ababc
s_2 = babc
s_3 = abc
s_4 = bc
s_5 = c



Suffix Tree

- Nodes that mark a split are called **essential**
- Remove non-essential nodes, and label edges with string



Questions

How is the tree created for ANA$?

Answer: Tree creation ANA$

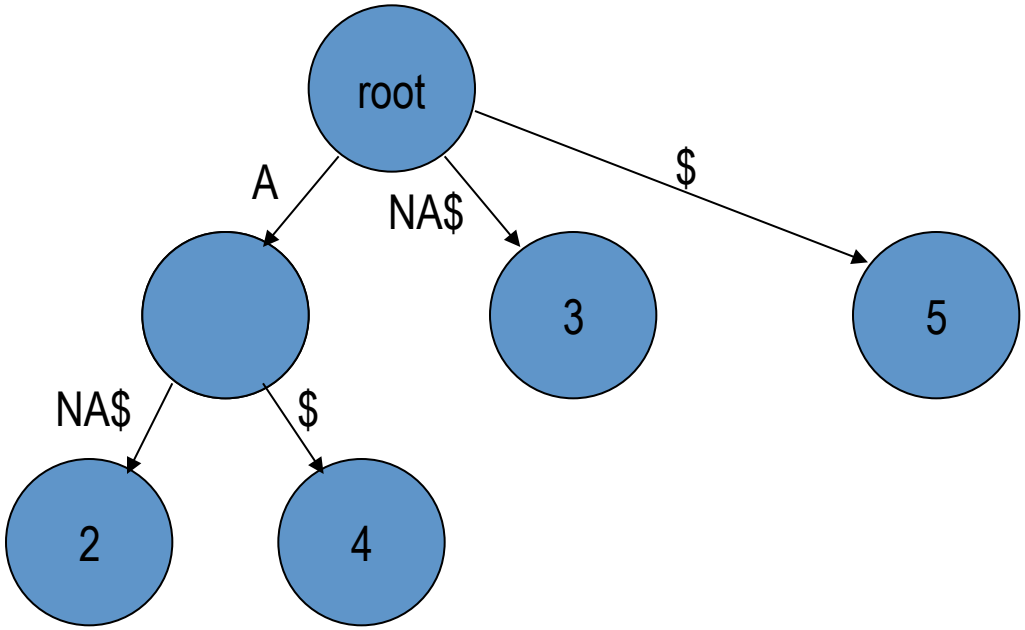
1 root

2 ANA$

3 NA$

4 A$

5 $

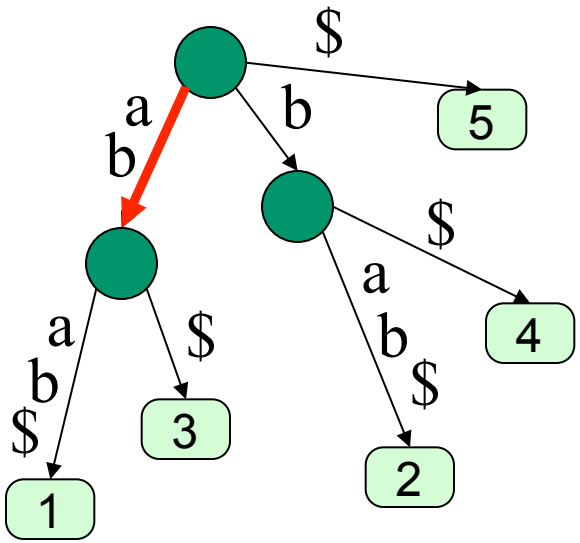


Questions

How is the tree created for CAGTCAGG$?

Exact string matching

In preprocessing we just build a suffix tree in O(n) time

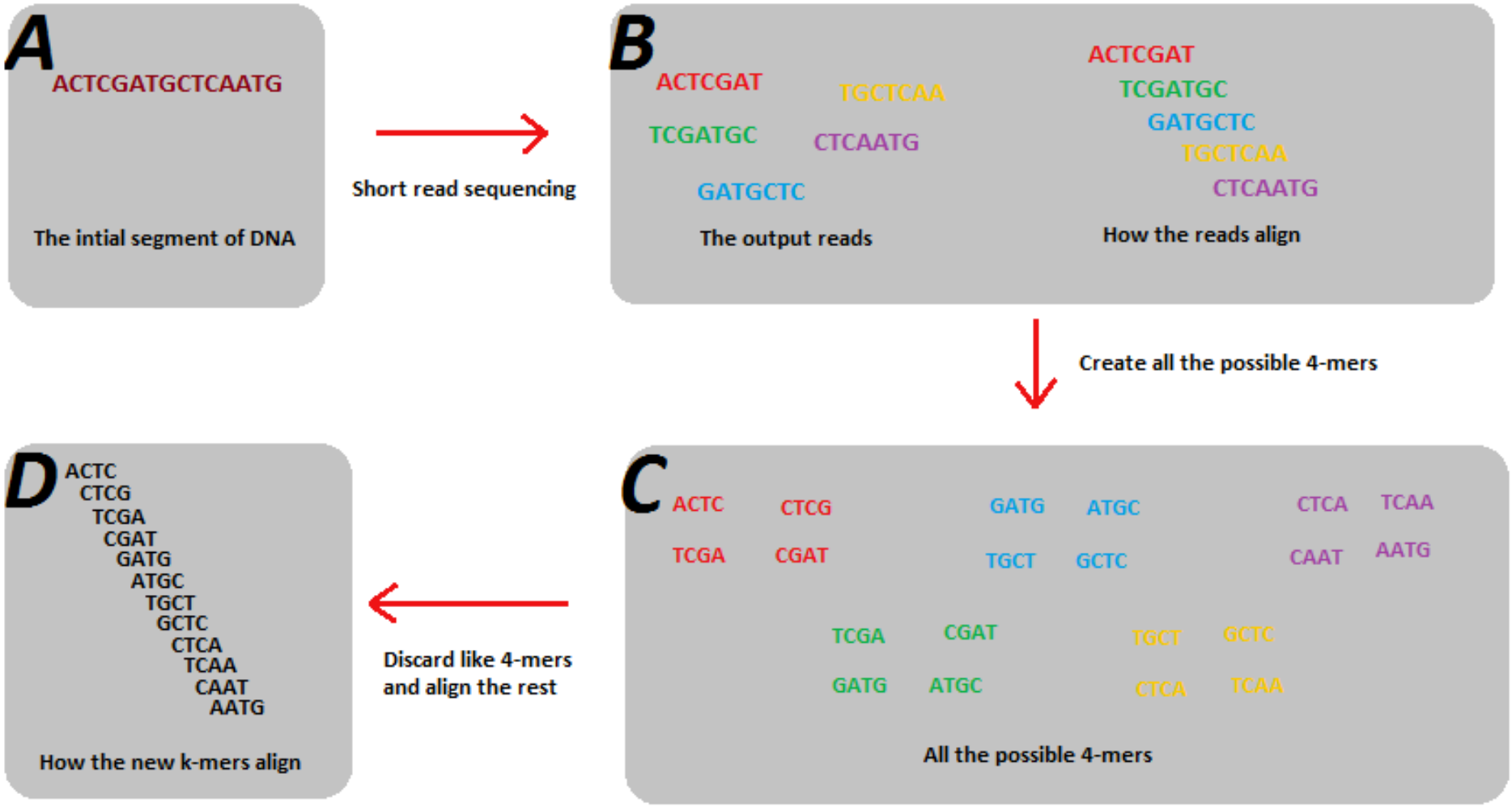


Given a pattern P = ab we traverse the tree according to the pattern.

Hashing: What is it?

- What does hashing do?
  - For different data, generate a unique integer
  - Store data in an array at the unique integer index generated from the data
- Hashing is a very efficient way to store and retrieve data

K-mer approach



https://en.wikipedia.org/wiki/K-mer#/media/File:K-mer-example.png

Hashing: Definitions

- Hash table: array used in hashing
- Records: data stored in a *hash table*
- Keys: identifies sets of *records*
- Hash function: uses a *key* to generate an index to insert at in *hash table*
- Collision: when more than one record is mapped to the same index in the hash table

Hashing DNA sequences

- Each k-mer can be translated into a binary string (**A**, **T**, **C**, **G** can be represented as **00**, **01**, **10**, **11)**
- After assigning a unique integer per k-mer it is easy to get all start locations of each k-mer in a genome

Hashing: Maximal Repeats

- To find repeats in a genome:
  - For all k-mers in the genome, note the start position and the sequence
  - Generate a hash table index for each unique k-mer sequence
  - In each index of the hash table, store all genome start locations of the k-mer which generated that index
  - Extend k-mer repeats to maximal repeats