Strings, Matching Algorithms and Indexing

Dr. Jared Simpson
Ontario Institute for Cancer Research
&
Department of Computer Science
University of Toronto

Today's Lecture

- String definitions and terminology
- Exact matching algorithms
- Substring indexing
- Approximate Matching
- Paper reading/discussion

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. $|\epsilon| = 0$

* sometimes we'll consider |Σ| 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* occurs 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.

P: `word`

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

Alignment 1:
```
word
```

Alignment 2:
```
word
```

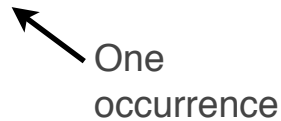
Exact matching

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 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

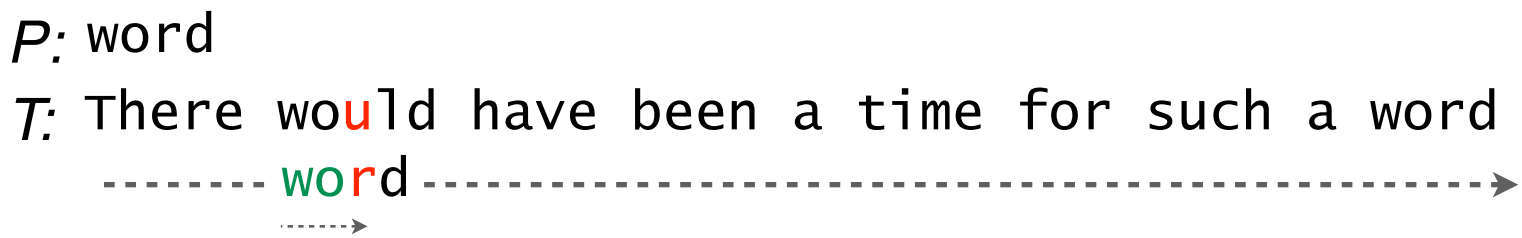
Exact matching: naïve algorithm summary

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 $\sim m$ character comparisons

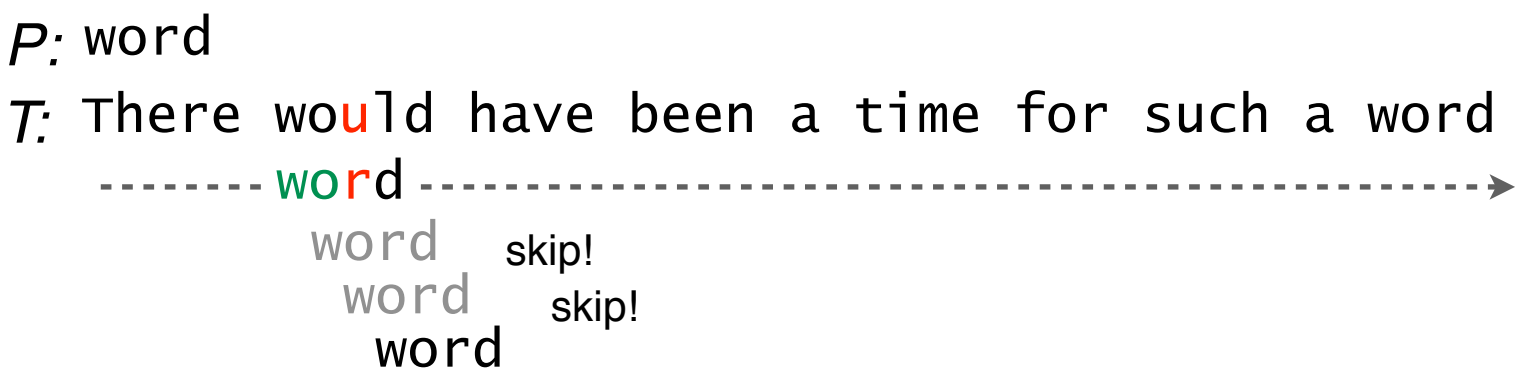
Exact matching: slightly less naïve algorithm



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

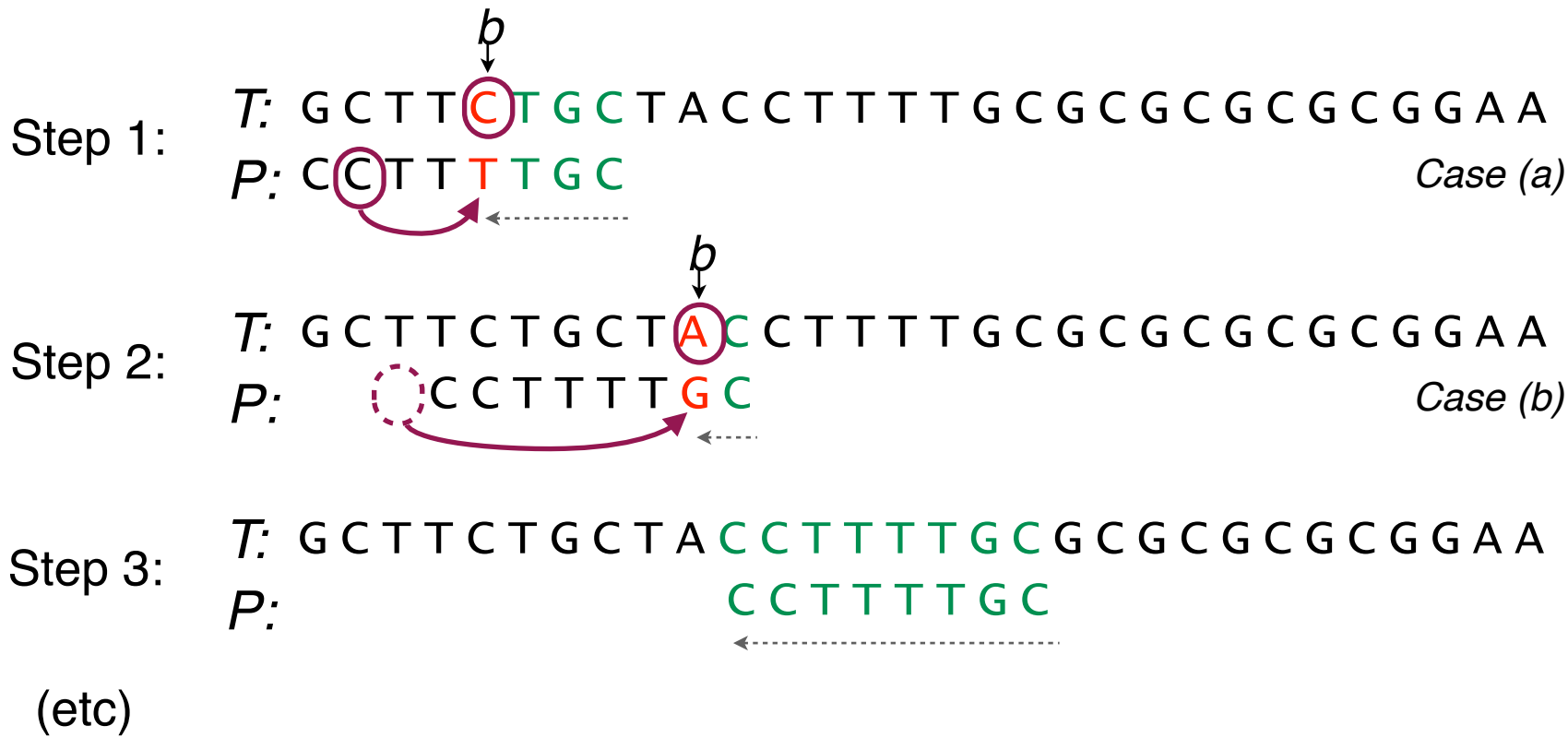
Use knowledge gained from character comparisons to skip future alignments that definitely won't match:

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

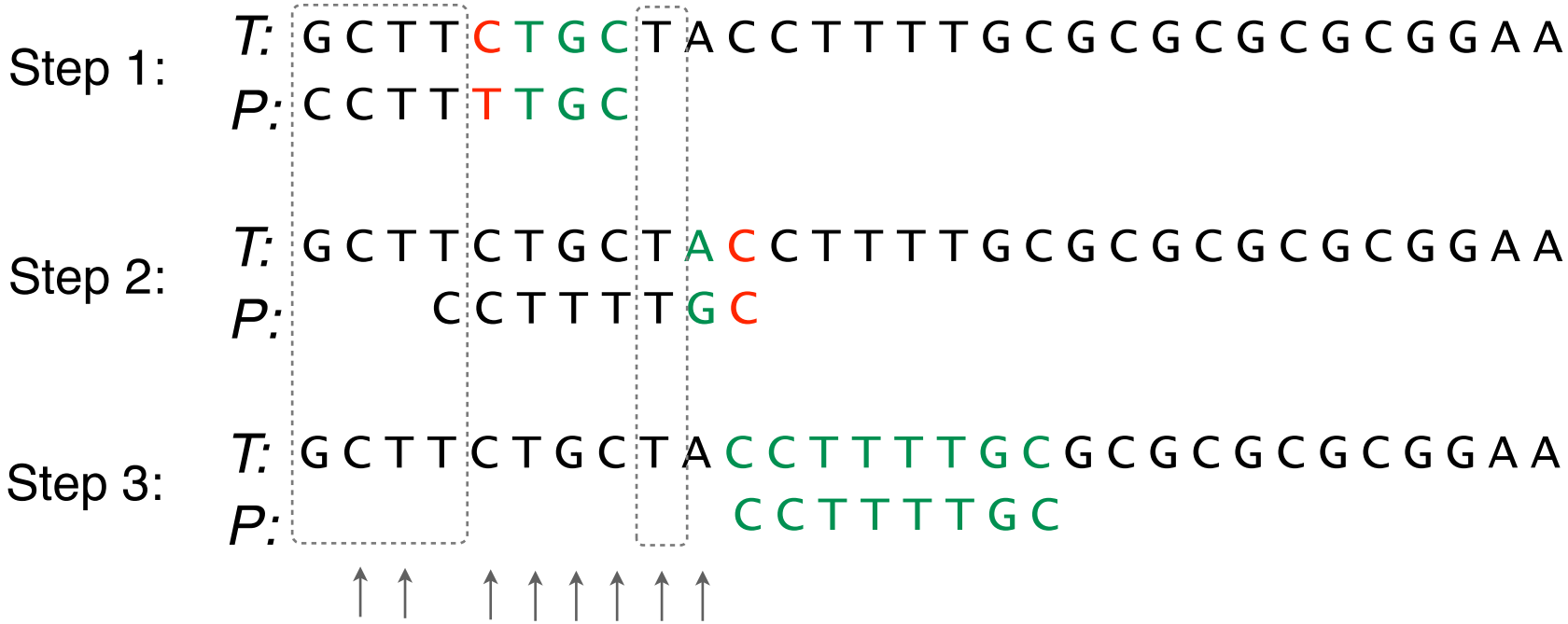
Boyer, RS and Moore, JS. "A fast string searching algorithm." *Communications of the ACM* 20.10 (1977): 762-772.

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



We skipped 8 alignments

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

Boyer-Moore: Bad character rule

As soon as *P* is known, build a | Σ |-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-Moore: Good suffix rule

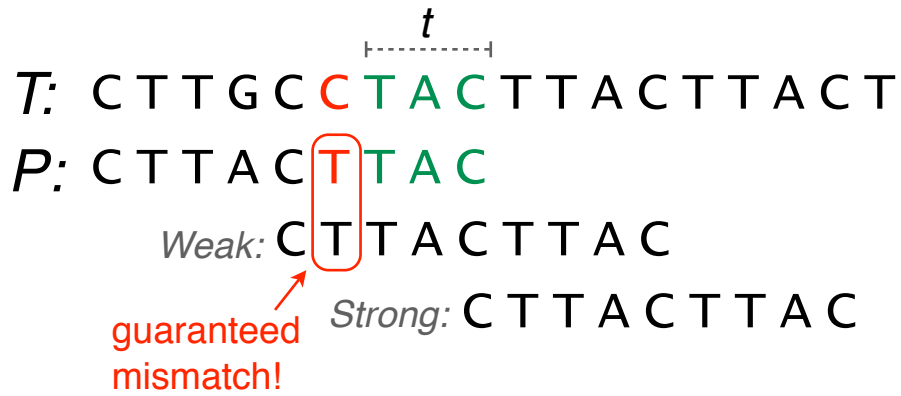
Let *t* be the substring of *T* that matched a suffix of *P*. Skip alignments until (a) *t* matches opposite characters in *P*, or (b) a prefix of *P* matches a suffix of *t*, or (c) *P* moves past *t,* whichever happens first

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

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)



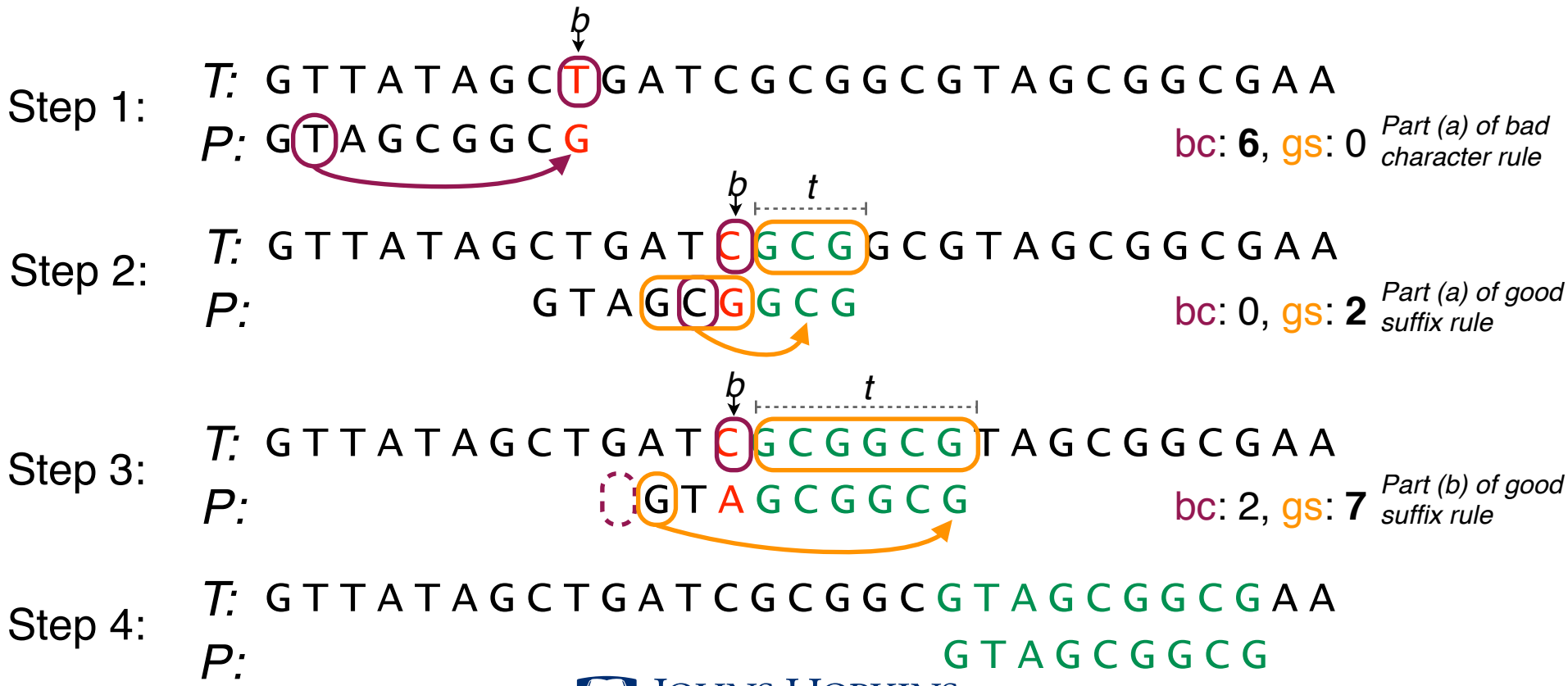
With the strong good suffix rule (and other minor modifications), Boyer-Moore is $O(m)$ *worst-case* time. Gusfield discusses proof.

Boyer-Moore: Putting it together

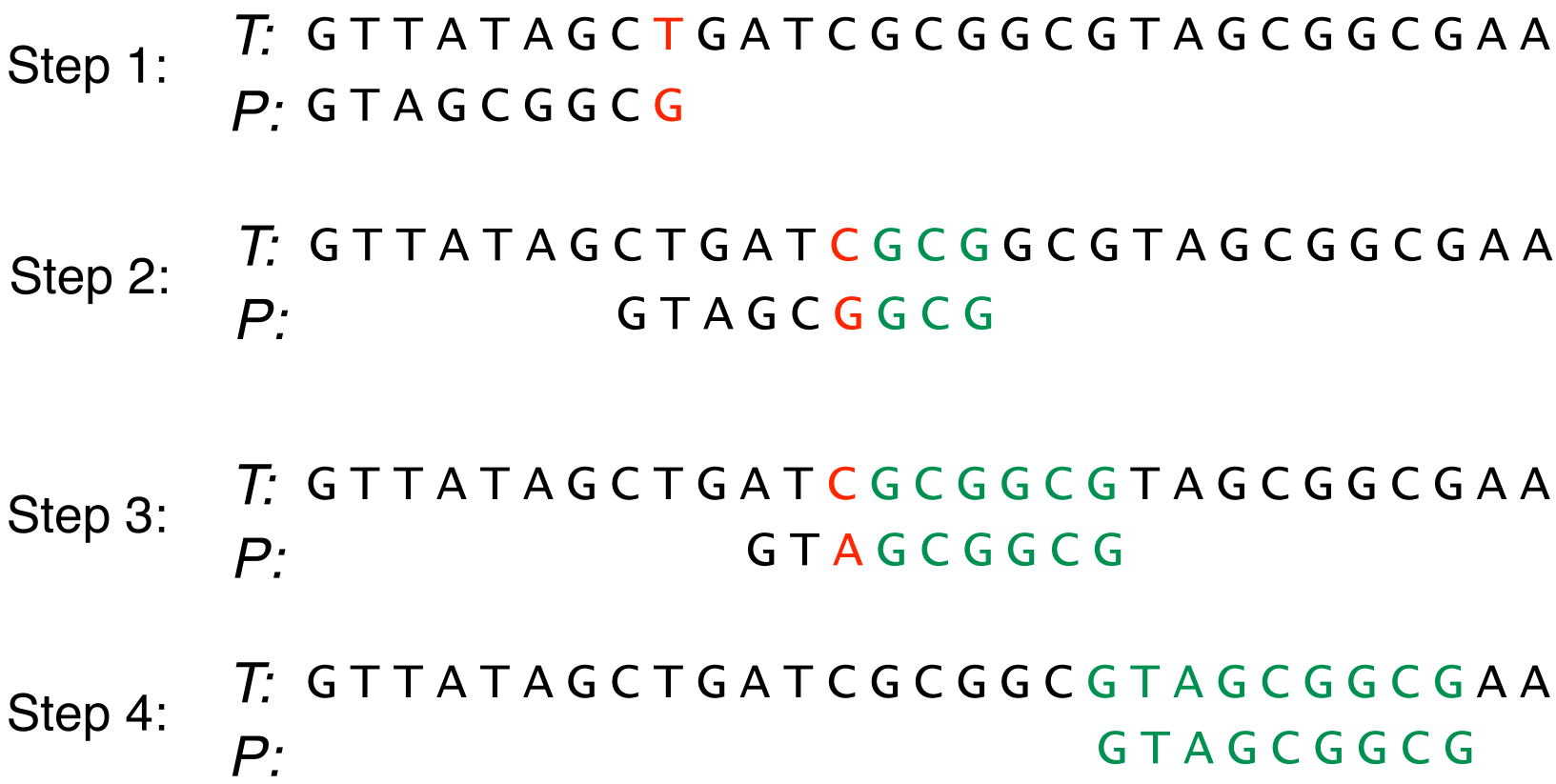
After each alignment, use bad character or good suffix rule, whichever skips more

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



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

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

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

Preprocessing

We saw the naive algorithm and Boyer-Moore. These and other algorithms can be distinguished by the kind of *preprocessing* they do.

Naive algorithm does no preprocessing

- Nothing for algorithm to do until it is given both pattern P and text T

Boyer-Moore preprocesses the pattern P

- If P is provided first, we can build lookup tables for the bad character and good suffix rules
- If T_1 is provided later on, we use the already-built tables to match P to T_1
- ...

Indexing

Can preprocess *P*, *T* or both. When preprocessing *T*, we say we are making an *index* of *T*, like the index of a book.

Sometimes called *inverted indexing,* since it inverts the word/page relationship

http://en.wikipedia.org/wiki/Index_(publishing)#mediaviewer/File:Atlas_maior_1655_-_vol_10_-_Novus_Atlas_Sinensis_-_index_-_P1080375.JPG

Web search engines also use inverted indexing

Documents:

```
T[0] = "it is what it is"
T[1] = "what is it"
T[2] = "it is a banana"
```

http://en.wikipedia.org/wiki/Inverted_index

Index:

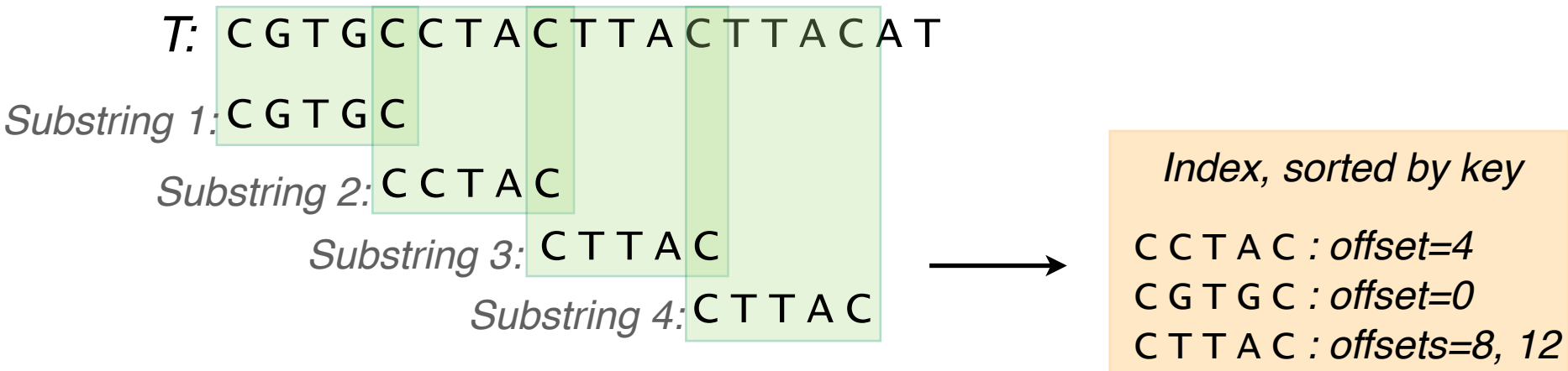
```
"a":      [2]
"banana": [2]
"is":     [0, 1, 2]
"it":     [0, 1, 2]
"what":   [0, 1]
```

Substring index

To index *T*:

Extract sequences from *T* (usually substrings), along with pointers back to where they occurred

Organize pieces and pointers into a *map* data structure. Various map structures are possible, all involving some mix of grouping / sorting.

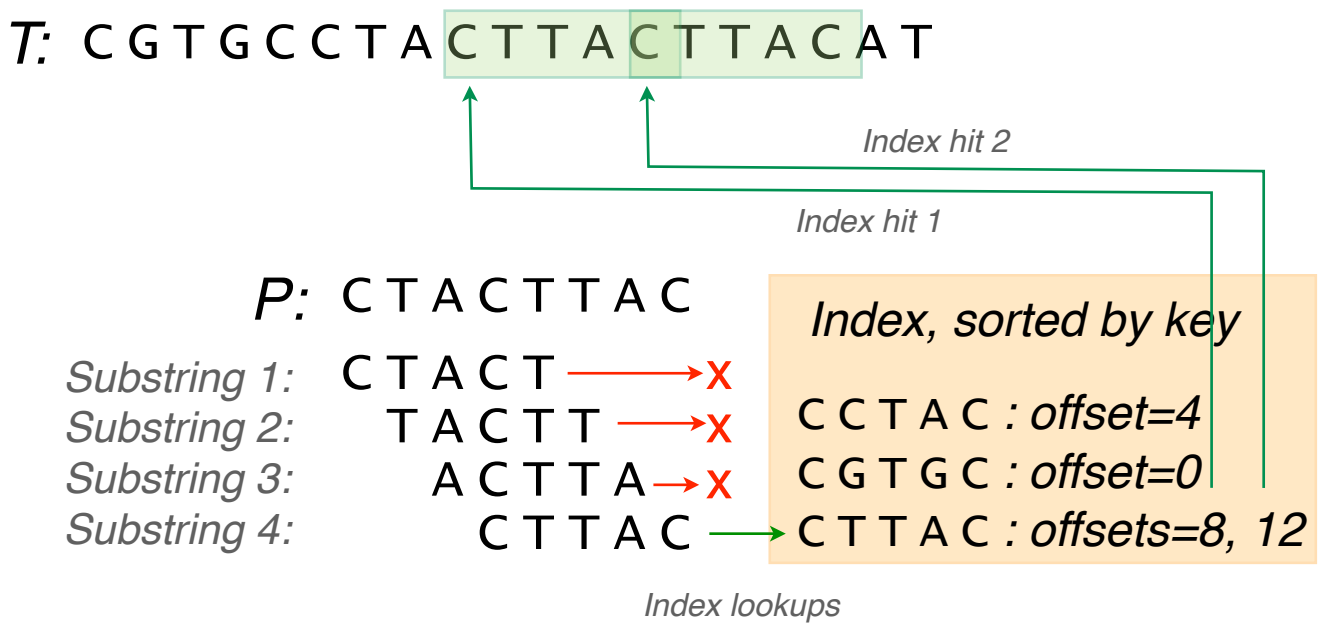


Substring index

To query index with *P*:

Extract substrings from *P*. Use them to query the index. Index tells us where they occur in *T*.

For each occurrence, check if there is a match in that vicinity



See Python example: http://nbviewer.ipython.org/6582444

Substring index

Index for *T*: sorted table of all length-2 substrings, and their offsets

To query: extract length-2 prefix of *P*, look up in index, investigate all hits

P: `ord`

Two hits to check:

T: `time_for_such_a_word`
P: `ord` `ord`

Hit at offset 6 but no match

Hit at offset 17 matches

T: `time_for_such_a_word`

Substring	Offset
_a	13
_f	4
_s	8
_w	15
a_	14
ch	11
e_	3
fo	5
h_	12
im	1
me	2
or	6
or	17
r_	7
rd	18
su	9
ti	0
uc	10
wo	16

Substring index: one implementation

```
import bisect
import sys

class Index(object):

    def __init__(self, t, ln=2):
        """ Create index, extracting substrings of length 'ln' """
        self.ln = ln
        self.index = []
        for i in xrange(0, len(t)-ln+1):
            self.index.append((t[i:i+ln], i)) # add <substr, offset> pair
        self.index.sort() # sort pairs

    def query(self, p):
        """ Return candidate alignments for p """
        st = bisect.bisect_left(self.index, (p[:self.ln], -1))
        en = bisect.bisect_right(self.index, (p[:self.ln], sys.maxint))
        hits = self.index[st:en]
        return [ h[1] for h in hits ] # return just the offsets

def queryIndex(p, t, index):
    """ Look for occurrences of p in t with help of index """
   ln = index.ln
    occurrences = []
    for i in index.query(p):
        if t[i+ln:i+len(p)] == p[ln:]:
            occurrences.append(i)
    return occurrences
```

`bisect` module implements binary search for us

Extract <substring, offset> pairs, put in list, sort list

Binary search for first & last entries matching substring

Get index hits, check for complete matches

```
>>> t = "time for such a word"
>>> ind = Index(t, ln=2, interval=2)
>>> queryIndex("ord", t, ind)
[17]
```

Substring index: comparison, part 1

Comparing simple Python implementations Boyer-Moore exact matching and an index like on previous slide, using length-4 substrings:

	Boyer-Moore		Sorted index of length-4 substrings				
	# character comparisons	wall clock time	# character comparisons	# index hits	wall clock time (query)	wall clock time (indexing)	
P: "tomorrow" **T**: Shakespeare's complete works	785,855	1.91s	68	17	0.00 s	10.22 s	17 matches $\|T\|$ = 5.59 M
P: 50 nt string from Alu repeat* **T**: Human reference (hg19) chromosome 1	32,495,111	67.21 s	*Very slow, took >12 GB of memory*				336 matches $\|T\|$ = 249 M

* GCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGG

Substring index: every other substring

Can we fix the large memory footprint?

Idea: instead of extracting every length-2 substring, skip some. E.g. skip every other.

To query *T*: extract leftmost 2 length-2 substrings of *P*, look up in index, try all candidates. First lookup will find matches at even offsets, second at odd offsets.

T: `time_for_such_a_word`

Index'(T):

Substring	Offset
_f	4
_s	8
a_	14
h_	12
me	2
or	6
rd	18
ti	0
uc	10
wo	16

Substrings in Index(T):

```
me fo _s ch a_ or
im _f r_ uc _a wo
ti e_ or su h_ _w rd
```

Pieces of P used to query Index(T): `or` ← all hits

T: `time_for_such_a_word`

P: `ord`

Substrings in Index'(T):

```
ti  _f  _s  h_  wo
 me  or  uc  a_  rd
```

Pieces of P used to query Index'(T):

```
or  ← just even
rd  ← just odd
```

Substring index: every *N*-th substring

...and just as we can take every other substring, we can also take every 3rd, 4th, 5th, etc

If we take every *N*-th substring, we must use each of the first *N* substrings of P to query the index

First query finds index hits corresponding to matches at offsets $\equiv 0 \pmod{N}$, second finds hits corresponding to match offsets $\equiv 1 \pmod{N}$, etc.

We'll call *N* the substring *interval*

Substring index: new implementation

```
import bisect
import sys

class Index2(object):

    def __init__(self, t, ln=2, interval=2):
        """ Create index, extracting substrings of length 'ln' every
            'interval' positions """
        self.ln = ln
        self.interval = interval
        self.index = []
        for i in xrange(0, len(t)-ln+1, interval):
            self.index.append((t[i:i+ln], i)) # add <substr, offset> pair
        self.index.sort() # sort pairs

    def query(self, p):
        """ Return candidate alignments for p """
        st = bisect.bisect_left(self.index, (p[:self.ln], -1))
        en = bisect.bisect_right(self.index, (p[:self.ln], sys.maxint))
        hits = self.index[st:en]
        return [ h[1] for h in hits ] # return just the offsets

def queryIndex2(p, t, index):
    """ Look for occurrences of p in t with help of index """
    ln, interval = index.ln, index.interval
    occurrences = []
    for k in xrange(0, interval): # For each offset into interval
        for i in index.query(p[k:]): # For each index hit
            # Test for match
            if t[i-k:i] == p[:k] and t[i+ln:i-k+len(p)] == p[k+ln:]:
                occurrences.append(i-k)
    return sorted(occurrences)
```

Loop stride

Configurable "interval" between substrings extracted from reference

When interval = *x*, extract first *x* substrings from *P* and do lookup for each

```
>>> t = "time for such a word"
>>> ind = Index2(t, ln=2, interval=2)
>>> queryIndex2("ord", t, ind)
[17]
```

Substring index: comparison, part 2

Comparing simple Python implementations Boyer-Moore exact matching and an index like on previous slide, using length-4 substrings extracted every 4 positions of *T*:

	Boyer-Moore		Sorted index of length-4 substrings, **interval=4**					
	# character comparisons	wall clock time	# character comparisons	# index hits	wall clock time (query)	wall clock time (indexing)	Peak memory footprint	
P: "tomorrow" **T**: Shakespeare's complete works	786 K	1.91s	34	17	0.00 s	13.69 s	150 MB	17 matches | *T* | = 5.59 M
P: 50 nt string from Alu repeat* **T**: Human reference (hg19) chromosome 1	32.5 M	67.21 s	445 K	277 K	0.40 s	59.31 s	7.6 GB	336 matches | *T* | = 249 M

Inverted indexing: map data structures

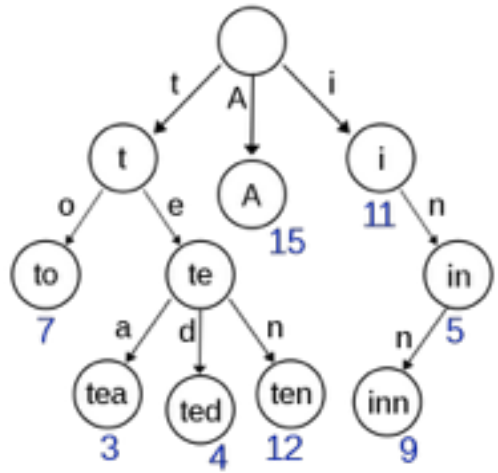
An inverted index maps substrings or subsequences to offsets where they occur

There are many candidate map data structures:

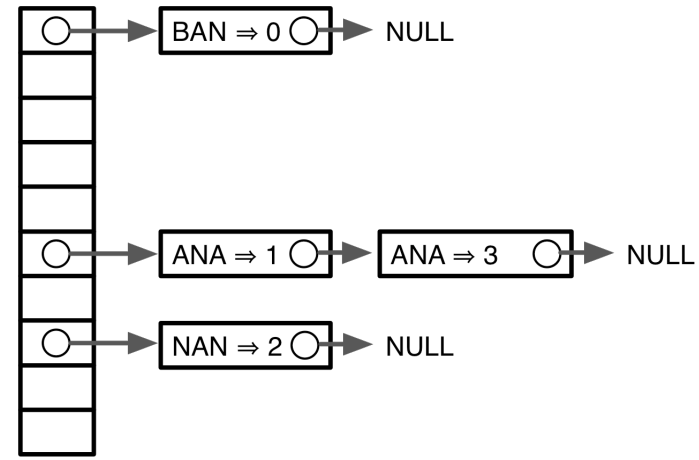
Sorted list:

_f	4
_s	8
a_	14
h_	12
me	2
or	6
rd	18
ti	0
uc	10
wo	16

Trie:

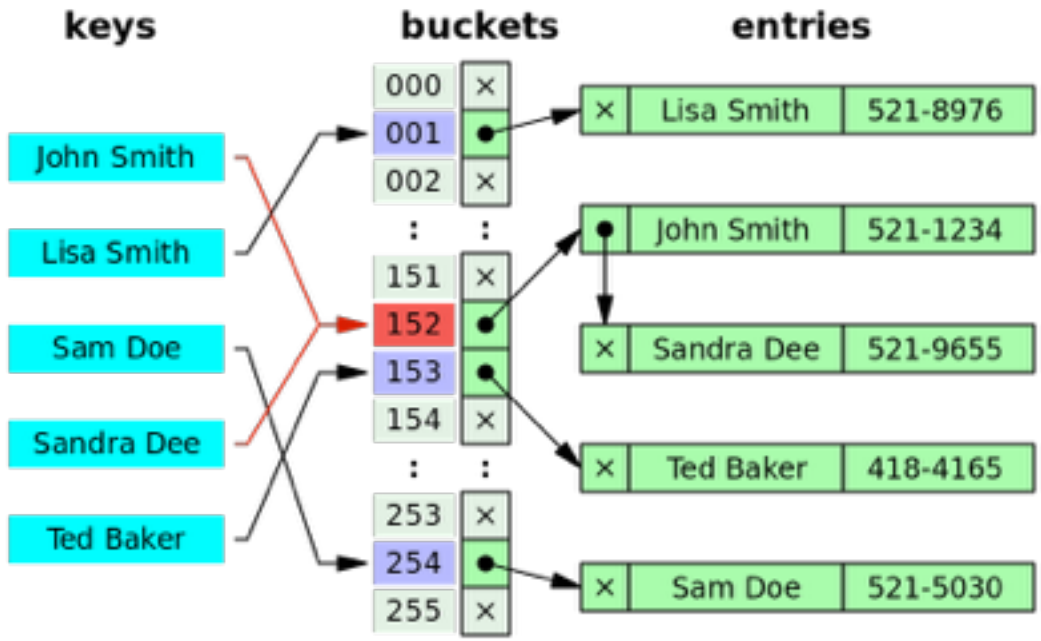


Hash table:



Indexing: quick hash table review

This hash table encodes a map from names to phone numbers:



Hash function maps a name to a *bucket id*: integers in [0, 256) in this case

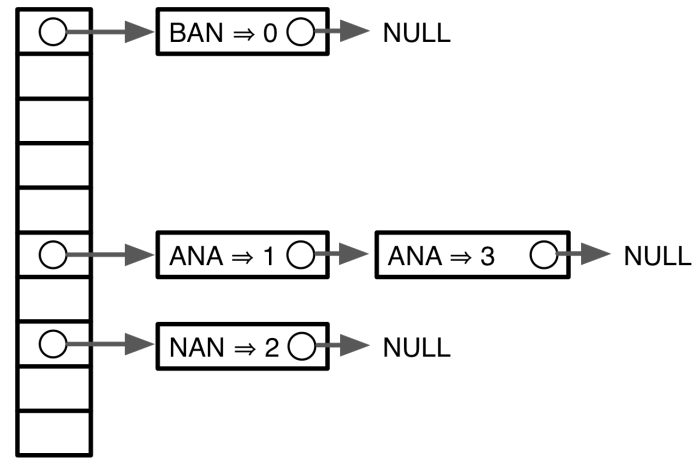
A bucket is a linked list of <key, value> pairs for keys that fell into that bucket

Involves grouping but no sorting. Many, many variations on this theme.

http://en.wikipedia.org/wiki/Hash_table

Indexing: hash versus sorted list

_f	4
_s	8
a_	14
h_	12
me	2
or	6
rd	18
ti	0
uc	10
wo	16



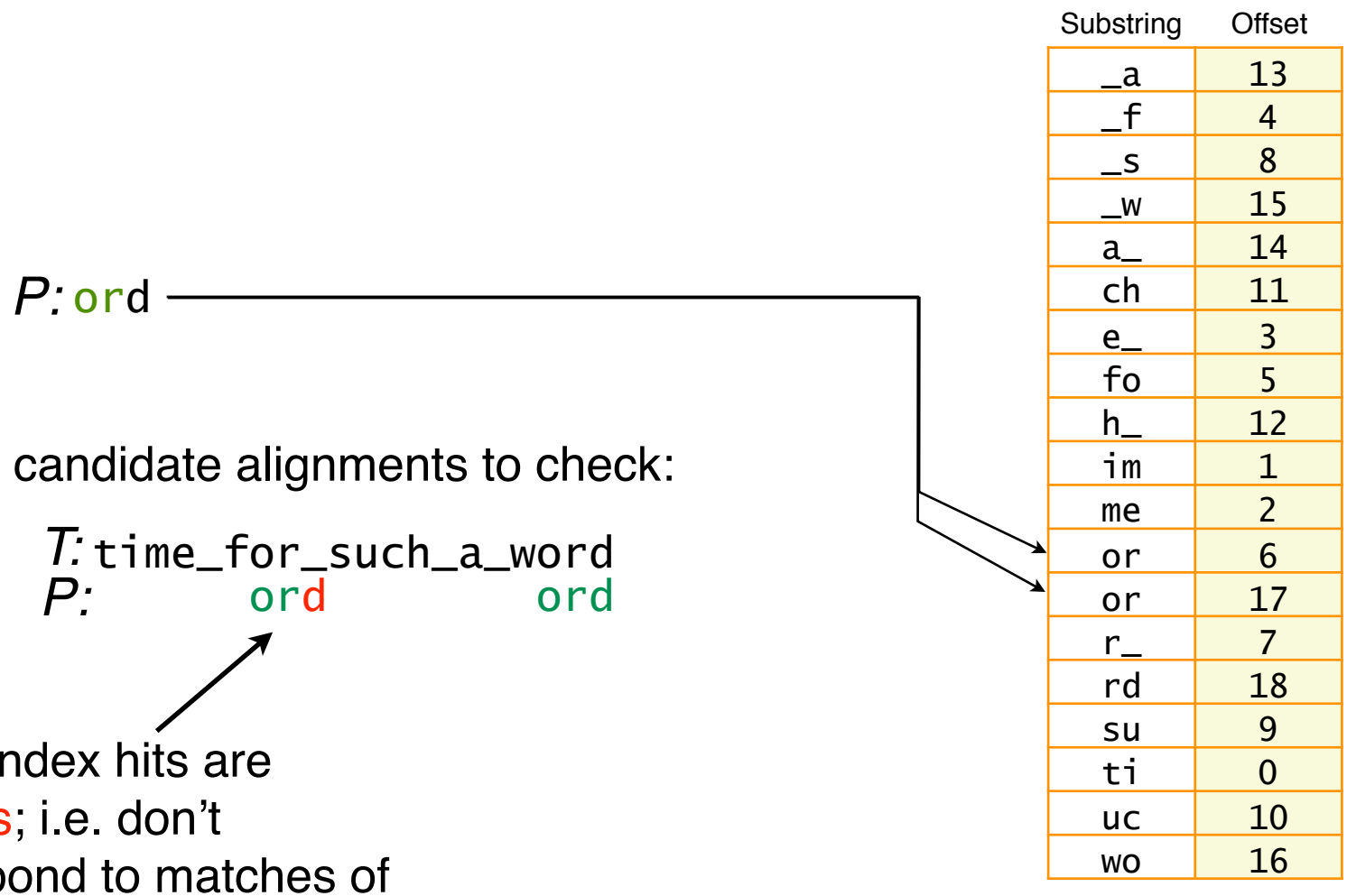
O(log *m*) query time

Just stores keys and values

O(1) query time on average

Must store keys, values, bucket array, pointers

Indexing: specificity



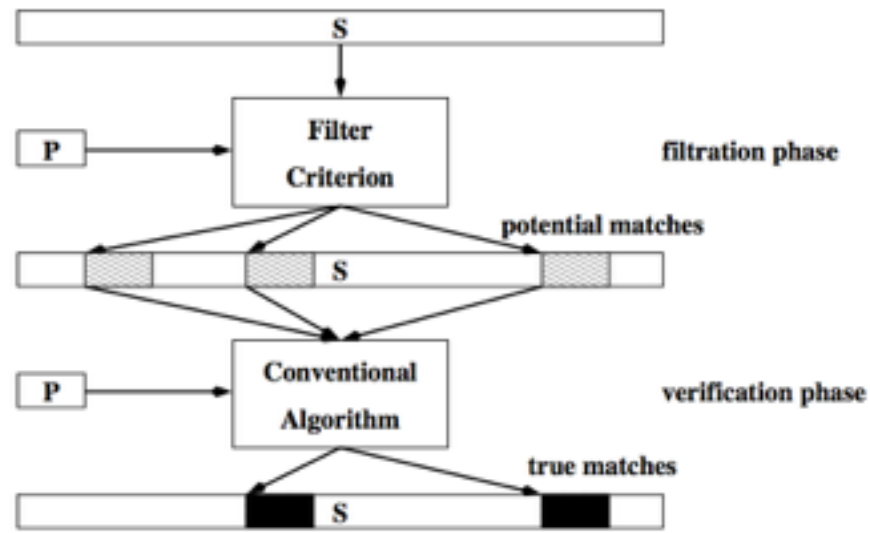
Indexing: specificity

Index-assisted method proceeds in two phases:

1. Index is queried to produce list of *candidate* loci (offsets)
2. Neighborhood around each candidate is checked for complete match

These are sometimes called *filter algorithms*, phase 1 being the filter

Specificity refers to the fraction of candidates from phase 1 that yield matches in phase 2. Higher specificity saves effort spent fruitlessly checking neighborhoods.



From: Burkhardt, Stefan. *Filter algorithms for approximate string matching*. Diss. Universitätsbibliothek, 2002.

Indexing: specificity comparison

Comparing specificities for several combinations of substring-length and interval settings. *P* & *T* are from the human chromosome 1 example.

Substring length	Interval	# character comparisons	# index hits	**specificity**	wall clock time (query)	wall clock time (indexing)	Peak memory footprint
4	4	445 K	277 K	0.12%	0.40 s	59.31 s	~7.6 GB
7	7	105 K	27 K	1.26%	0.06 s	63.74 s	~5.0 GB
10	10	62 K	8.2 K	4.11%	0.02 s	72.52 s	~3.6 GB
18	18	49 K	7.7 K	4.37%	0.02 s	40.20 s	~2.1 GB
30	30	13 K	1.9 K	11.72%	0.00 s	19.78 s	~1.6 GB

Increasing substring and interval lengths increases specificity, which improves query time on balance. In many cases, the increasing interval improves index size and building time.

Read alignment requires approximate matching

Read

CTCAAACTCCTGACCTTTGGTGATCCACCCGCCTNGGCCTTC

Reference

Sequence differences occur because of...

1. Sequencing error
2. Natural variation

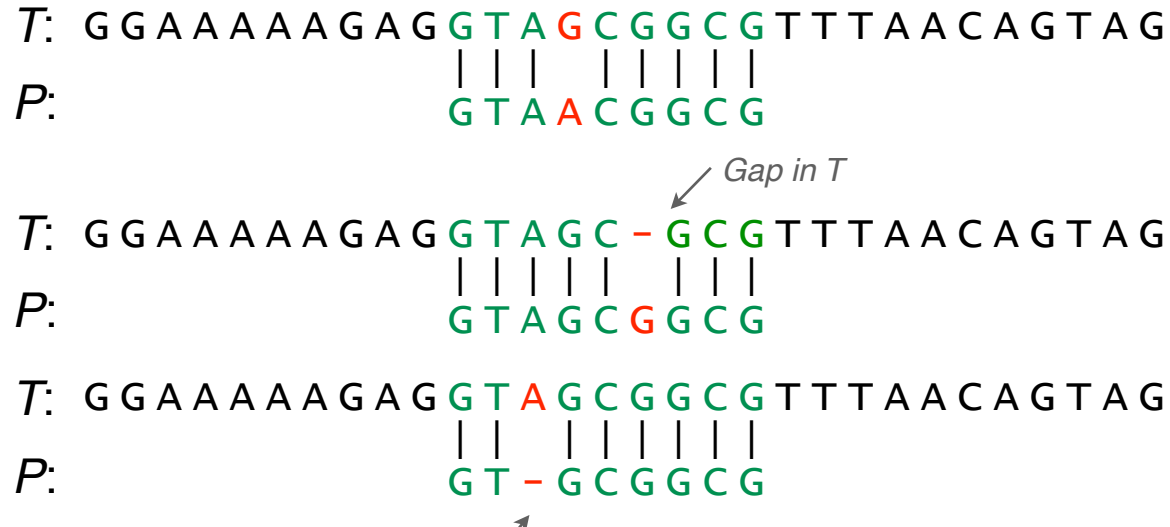
JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Approximate string matching

Looking for places where a *P* matches *T* with up to a certain number of mismatches or edits. Each such place is an *approximate match*.

A *mismatch* is a single-character substitution:

An *edit* is a single-character substitution or *gap* (*insertion* or *deletion*):



JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Hamming and edit distance

For two same-length strings *X* and *Y*, *hamming distance* is the minimum number of single-character substitutions needed to turn one into the other:

```
X: G A G G T A G C G G C G T T T A A C
   |   | | | |   | | |   | | | | | | |
Y: G T G G T A A C G G G G T T T A A C
```

Hamming distance = 3

Edit distance (*Levenshtein distance*): minimum number of *edits* required to turn one into the other:

```
X: T G G C C G C G C A A A A A C A G C
   | |   | | | | | | | | | |   | | | |
Y: T G A C C G C G C A A A A - C A G C
```

Edit distance = 2

```
X: G C G T A T G C G G C T A - A C G C
   | |   | | | | | | | | | |   | | |
Y: G C - T A T G C G G C T A T A C G C
```

Edit distance = 2

Approximate string matching

Adapting the naive algorithm to do approximate string matching within configurable Hamming distance:

```
def naiveApproximate(p, t, maxHammingDistance=1):
    occurrences = []
    for i in xrange(0, len(t) - len(p) + 1): # for all alignments
        nmm = 0
        for j in xrange(0, len(p)):          # for all characters
            if t[i+j] != p[j]:               # does it match?
                nmm += 1                     # mismatch
                if nmm > maxHammingDistance:
                    break                    # exceeded maximum distance
        if nmm <= maxHammingDistance:
            # approximate match; return pair where first element is the
            # offset of the match and second is the Hamming distance
            occurrences.append((i, nmm))
    return occurrences
```

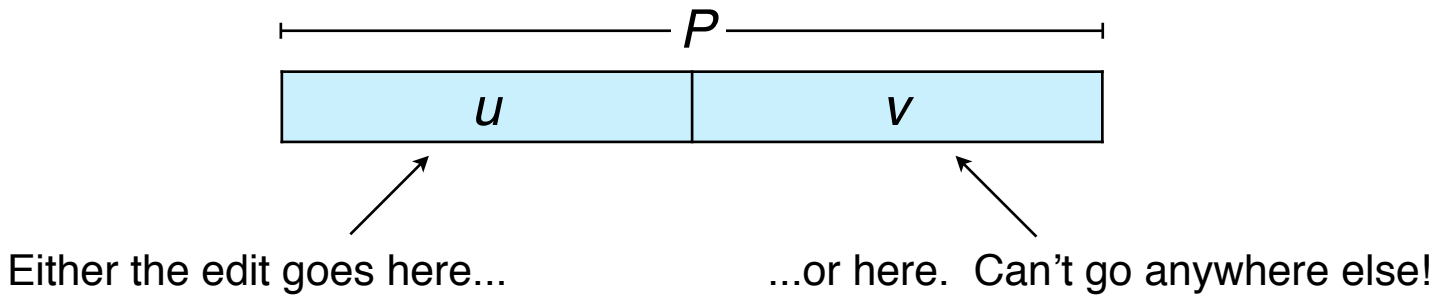
Instead of stopping upon first mismatch, stop when maximum distance is exceeded

Python example: http://bit.ly/CG_NaiveApprox

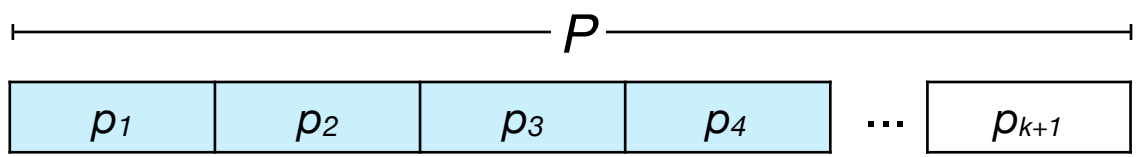
Approximate string matching

How to make Boyer-Moore and index-assisted exact matching approximate?

Helpful fact: Split *P* into non-empty non-overlapping substrings *u* and *v*. If *P* occurs in *T* with 1 edit, either *u* or *v* must match exactly.



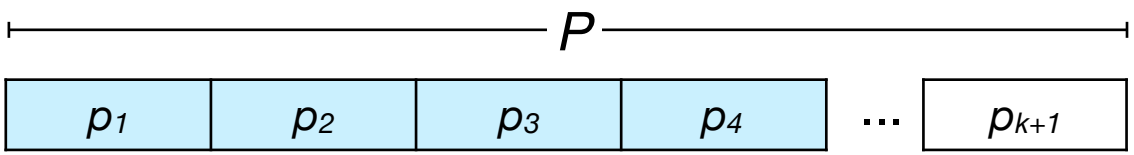
More generally: Let $p_1, p_2, ..., p_{k+1}$ be a partitioning of *P* into *k*+1 non-overlapping non-empty substrings. If *P* occurs in *T* with up to *k* edits, then at least one of $p_1, p_2, ..., p_{k+1}$ must match exactly.



≤ *k* edits can affect as many as *k* of these, but not all

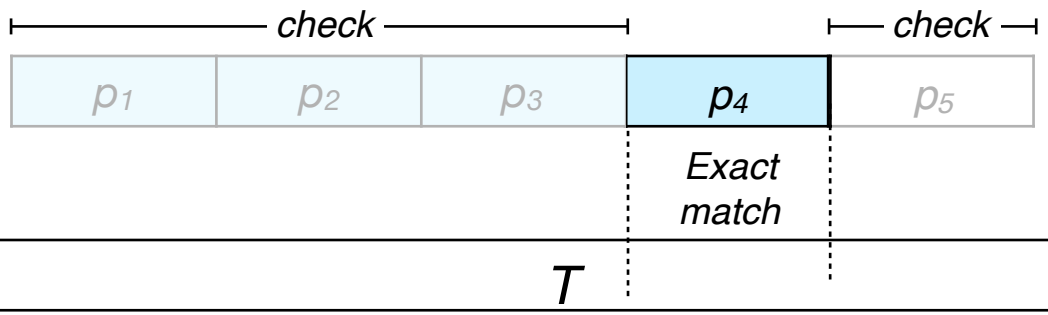
Approximate string matching

These rules provides a bridge from the exact-matching methods we've studied so far, and approximate string matching.



≤ *k* edits can overlap as many as *k* of these, but not all

Use an exact matching algorithm to find exact matches for $p_1, p_2, \ldots, p_{k+1}$. Look for a longer approximate match in the vicinity of the exact match.



Approximate string matching

```
def bmApproximate(p, t, k, alph="ACGT"):
    """ Use the pigeonhole principle together with Boyer-Moore to find
        approximate matches with up to a specified number of mismatches. """
    if len(p) < k+1:
        raise RuntimeError("Pattern too short (%d) for given k (%d)" % (len(p), k))
    ps = partition(p, k+1) # split p into list of k+1 non-empty, non-overlapping substrings
    off = 0                # offset into p of current partition
    occurrences = set()    # note we might see the same occurrence >1 time
    for pi in ps:          # for each partition
        bm_prep = BMPreprocessing(pi, alph=alph) # BM preprocess the partition
        for hit in bm_prep.match(t)[0]:
            if hit - off < 0: continue # pattern falls off left end of T?
            if hit + len(p) - off > len(t): continue # falls off right end?
            # Count mismatches to left and right of the matching partition
            nmm = 0
            for i in range(0, off) + range(off+len(pi), len(p)):
                if t[hit-off+i] != p[i]:
                    nmm += 1
                    if nmm > k: break # exceeded maximum # mismatches
            if nmm <= k:
                occurrences.add(hit-off) # approximate match
        off += len(pi) # Update offset of current partition
    return sorted(list(occurrences))
```

Full example: http://bit.ly/CG_BoyerMooreApprox

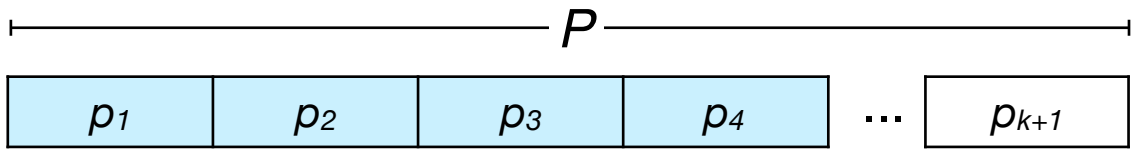
Approximate Boyer-Moore performance

	Boyer-Moore, exact			Boyer-Moore, ≤1 mismatch with pigeonhole			Boyer-Moore, ≤2 mismatches with pigeonhole		
	# character comparisons	wall clock time	# matches	# character comparisons	wall clock time	# matches	# character comparisons	wall clock time	# matches
P: "tomorrow" **T**: Shakespeare's complete works	786 K	1.91s	17	3.05 M	7.73 s	24	6.98 M	16.83 s	382
P: 50 nt string from Alu repeat* **T**: Human reference (hg19) chromosome 1	32.5 M	67.21 s	336	107 M	209 s	1,045	171 M	328 s	2,798

* GCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGG

Approximate string matching: more principles

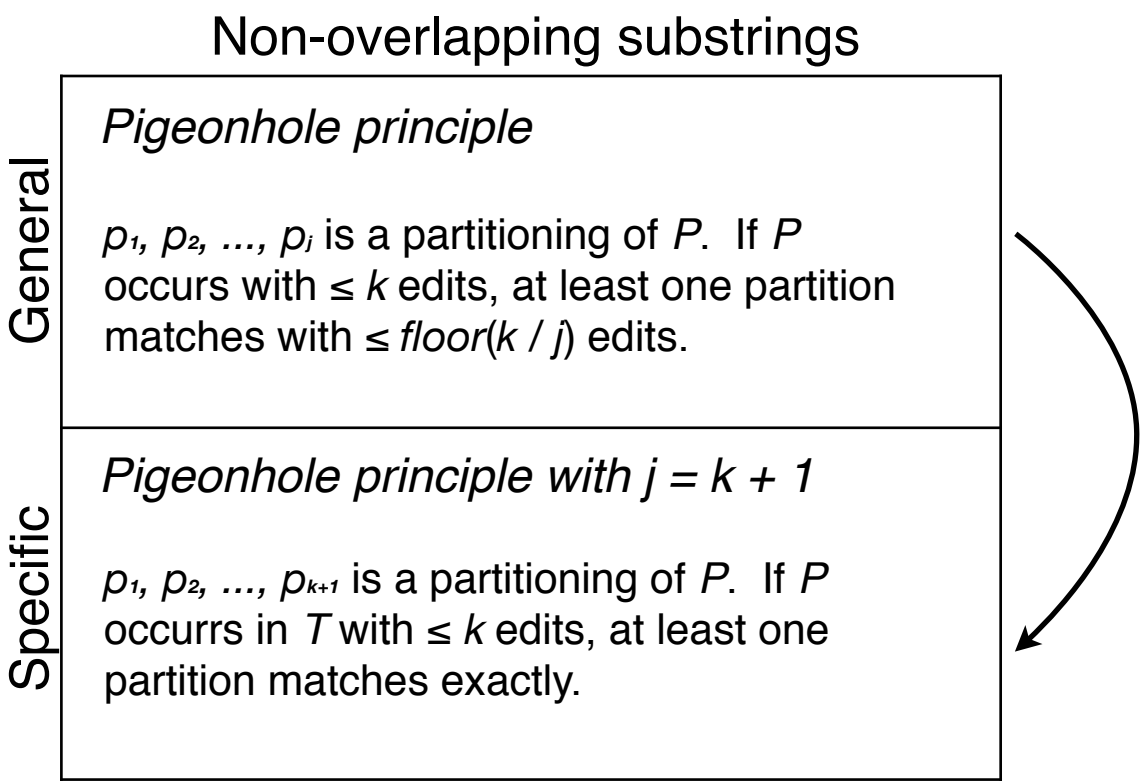
Let $p_1, p_2, ..., p_{k+1}$ be a partitioning of P into k+1 non-overlapping non-empty substrings. If P occurrs in T with up to k edits, then at least one of $p_1, p_2, ..., p_{k+1}$ must match exactly.



New principle:

Let $p_1, p_2, ..., p_j$ be a partitioning of P into j non-overlapping non-empty substrings. If P occurs with up to k edits, then at least one of $p_1, p_2, ..., p_j$ must occur with $\leq$ *floor*(k / j) edits.

Review: approximate matching principles



Let $j = k + 1$

Why?

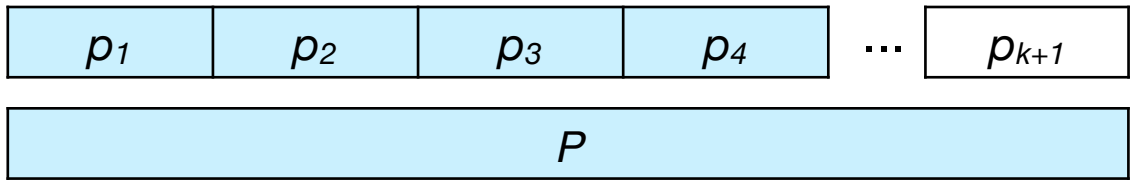
Smallest value s.t. $floor(k / j) = 0$

Why make $floor(k / j) = 0$?

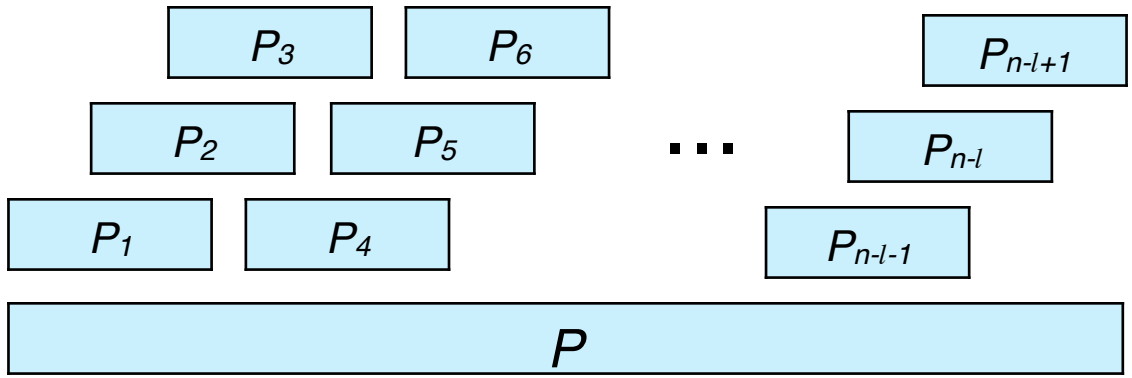
So we can use exact matching

Approximate string matching: more principles

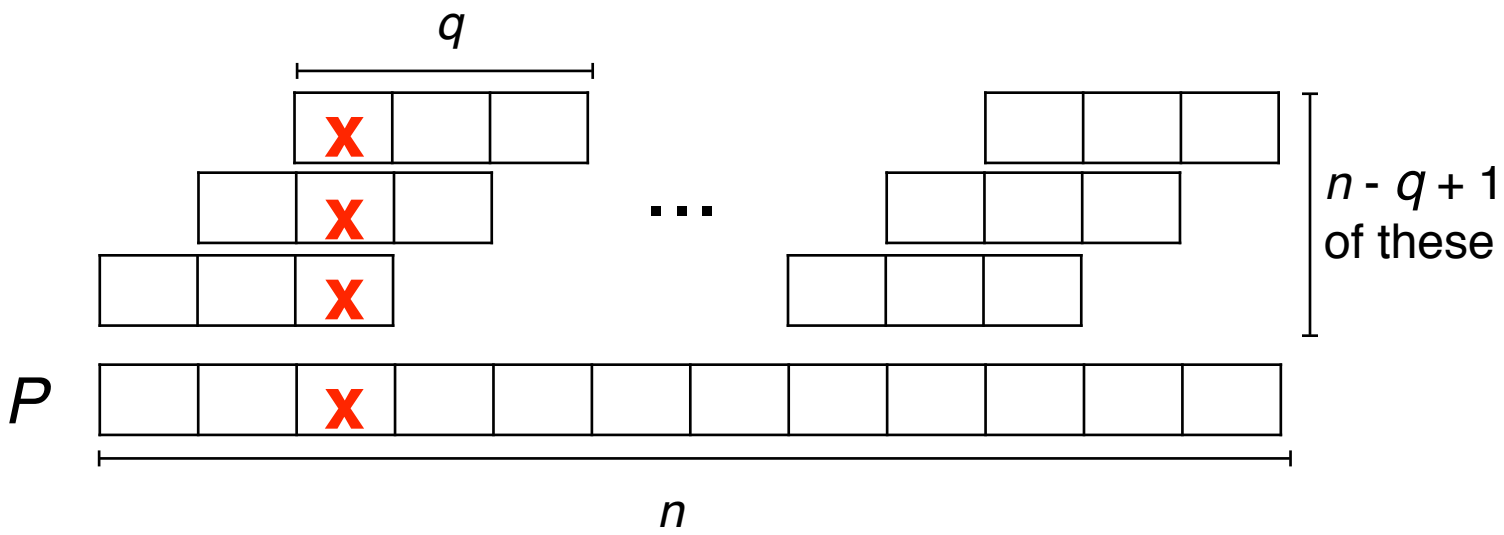
We partitioned P into non-overlapping substrings



Consider overlapping substrings



Approximate string matching: more principles



Say substrings are length *q*. There are *n* - *q* + 1 such substrings.

Worst case: 1 edit to *P* changes up to *q* substrings

Minimum # of length-*q* substrings unedited after *k* edits? $n - q + 1 - kq$

q-gram lemma: if *P* occurs in *T* with up to *k* edits, alignment must contain *t* exact matches of length *q*, where $t \geq n - q + 1 - kq$

Approximate string matching: more principles

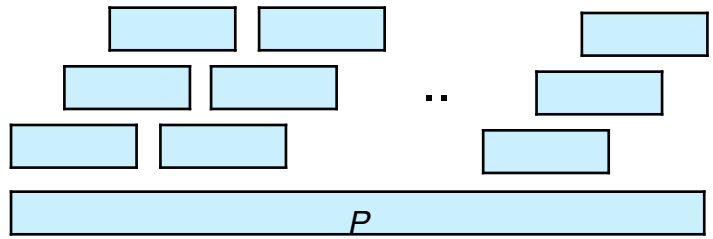
If P occurs in T with up to k edits, alignment contains an exact match of length q, where $q \geq floor(n / (k + 1))$

Derived by solving this for q: $n - q + 1 - kq \geq 1$

Exact matching filter: find matches of length $floor(n / (k + 1))$ between T and *any* substring of P. Check vicinity for full match.

Approximate matching principles

	Non-overlapping substrings	Overlapping substrings
General	*Pigeonhole principle* $p_1, p_2, \ldots, p_j$ is a partitioning of P. If P occurs with ≤ k edits, at least one partition matches with ≤ $floor(k / j)$ edits.	*q-gram lemma* If P occurs with ≤ k edits, alignment contains t exact matches of length q, where $t \geq n - q + 1 - kq$
Specific	*Pigeonhole principle with j = k + 1* $p_1, p_2, \ldots, p_{k+1}$ is a partitioning of P. If P occurrs in T with ≤ k edits, at least one partition matches exactly.	*q-gram lemma with t = 1* If P occurs with ≤ k edits, alignment contains an exact match of length q where $q \geq floor(n / (k + 1))$

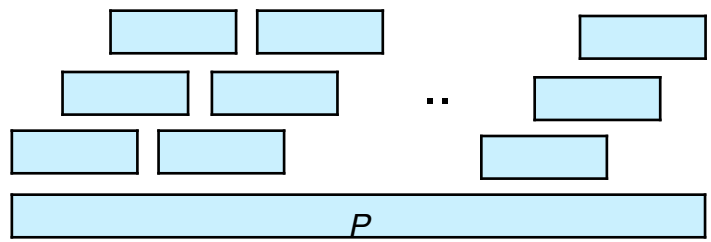


Sensitivity

Sensitivity = fraction of “true” approximate matches discovered by the algorithm

Lossless algorithm finds all of them, *lossy* algorithm doesn’t necessarily

We’ve seen *lossless* algorithms. Most everyday tools are *lossy.* Lossy algorithms are often much speedier & still acceptably sensitive (e.g. BLAST, BLAT, Bowtie).



Example lossy algorithm: pick $q > floor(n / (k + 1))$

Paper discussion

- Time permitting we will read these papers to illustrate these concepts:
  - “SSAHA: A Fast Search Method for Large DNA Databases” - Ning et al
    - example of software using non-overlapping substrings
  - “Efficient q-Gram filters for finding all ϵ-matches over a given length” - Rasmussen et al.
    - Example of overlapping substrings
    - More complicated strategy than SSAHA
    - Focus on understanding the problem (section 1) and the relationship of their strategy to other work (related work/contributions section)

Questions for discussion

- SSAHA paper
  - What is the indexing strategy?
  - How is it implemented?
  - What substring length do they use?
  - Why do they represent the strings as integers?
  - They ignore substrings that occur more than some number of times, N. Why do they do this and what is the downside?
- q-gram paper
  - Why do they use overlapping q-grams rather than non-overlapping like SSAHA?
  - What is an ϵ-match?