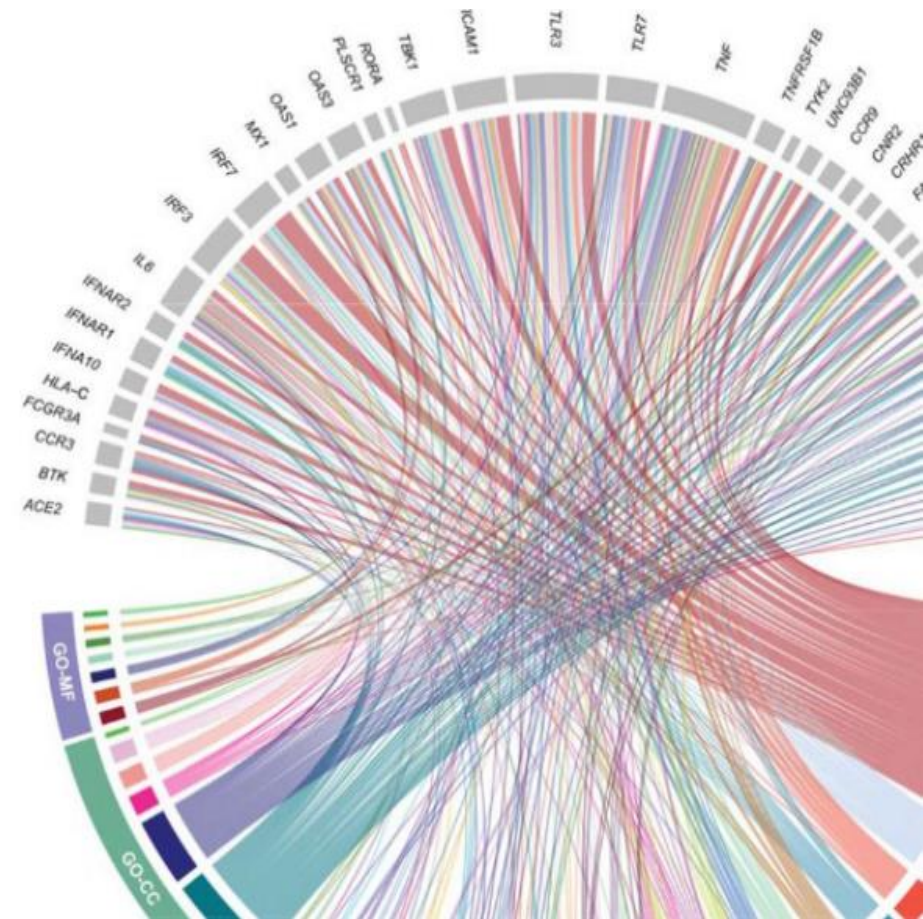


Tècniques i Eines Bioinformàtiques

Approximate String Matching

Santiago Marco-Sola (santiago.marco@upc.edu)

*Màster en Enginyeria Informàtica, UPC
Departament of Computer Science
Facultat d'Informàtica de Barcelona (FIB), UPC*



Acknowledgements

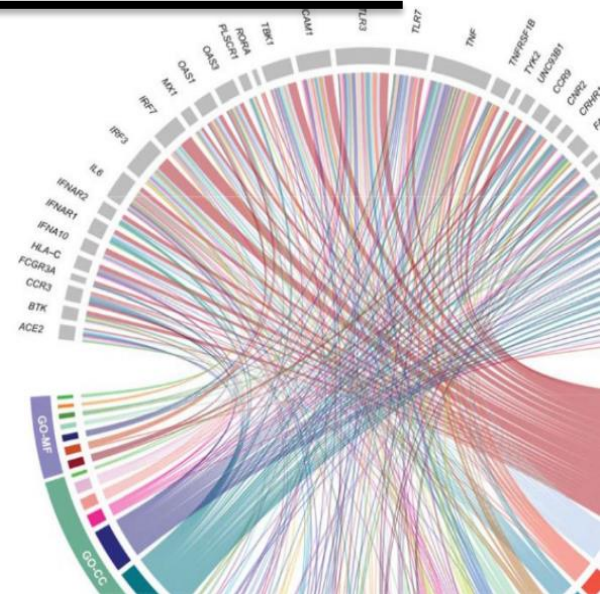
Many pictures and materials are taken from **Ben Langmead's course**.

Course heavily inspired in:

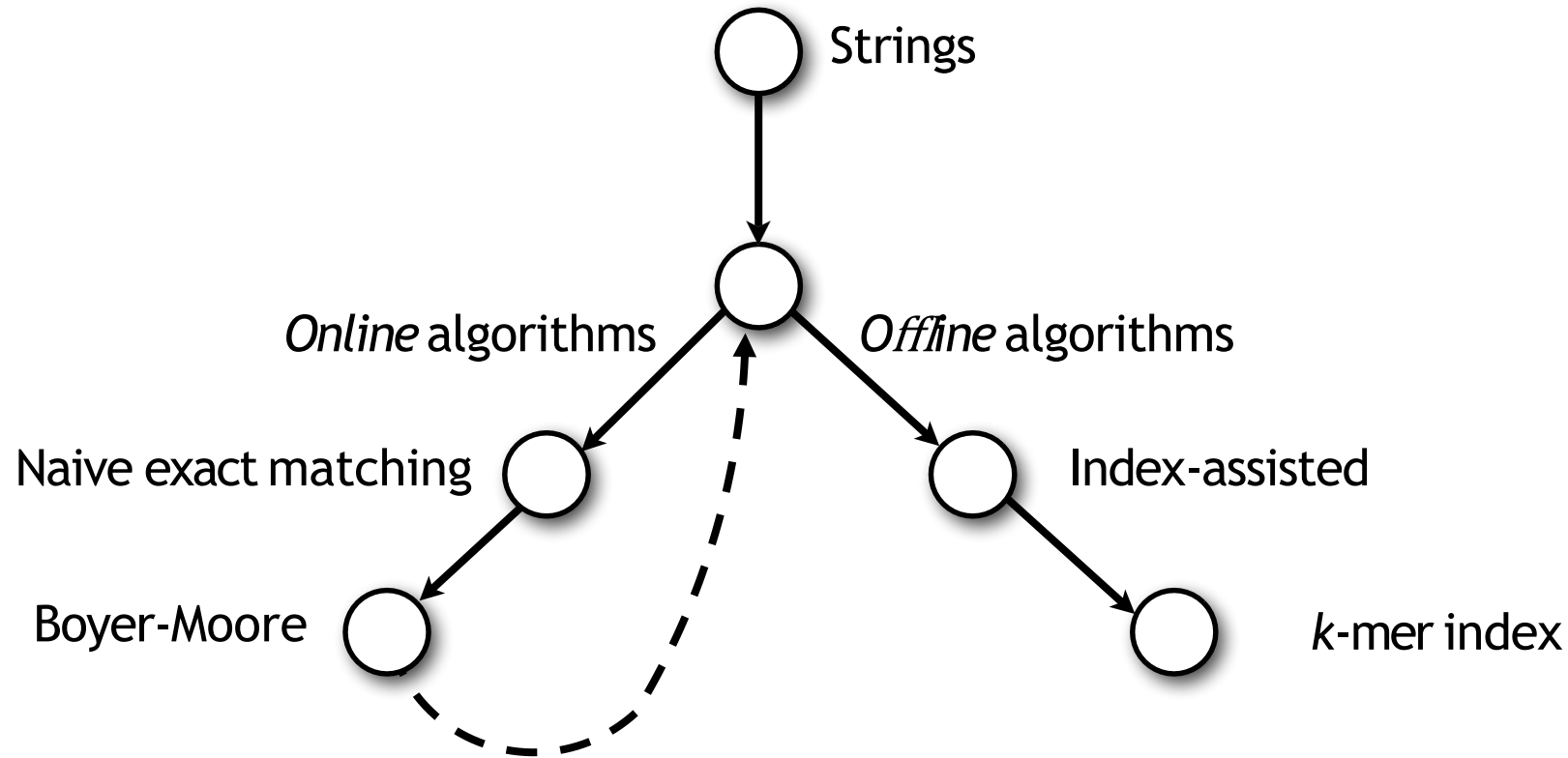
- **Genome-Scale Algorithm Design.** Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, Alexandru I. Tomescu. Cambridge University Press.
- **Algorithms on Strings, Trees, and Sequences.** Dan Gusfield. Cambridge University Press.
- **An Introduction to Bioinformatics Algorithms.** Neil C. Jones, Pavel A. Pevzner. MIT Press.

1

Approximate String Matching



Approximate matching



We have focused on *exact* matching...
... in reality, we have to deal with *differences*

Reference

GATCACAGGTCTATCACCTTATTAACCACTCACGGGAGCTCTCCATGCATTTGGTATTTT
CGTCTGGGGGGTATGCACGCGATAGCATTGCGAGACGCTGGAGCCGGAGCACCTATGTC
GCAGTATCTGTCTTTGATTCTGCCTCATCTATTATTTATCGCACCTACGTTCAATATT
ACAGGCGAACATACTTACTAAAGTGTGTTAATTAATTAATGCTTGTAGGACATAATAATA
ACAATTGAATGTCTGCACAGCCACTTTCCACACAGACATCATACAAAAAATTTCCACCA
AACCCCCCTCCCCGCTTCTG GCCACAGCCTTAAACCTCTCTGCCAAACCCCAAAA
ACAAAGAACCCTAACACCAGCCTAACCAATTTCAAATTTTCTCTGGCGGTATGCAC
TTTAAACAGTCACCCCCCAACTAACCAATTTTCCCCTCCCACTTCATACCTACTAAT
CTCATCAATACAACCCCGCCATCTTACCCAGCACACACACACCTTCTAACCCATA
CCCCGAACCAACCAACCCCAAACCTCACCCCCCAGTTTTATGTAGCTTCTCTCTCAA
GCAATACACTGACCCGCTCAAACTCTGGATTCTTGGATCCACCCAGCGCTTGGCCTAAA
CTAGCCTTTCTATTAGCTCTTAGAAGATTACACATGCAAGCATCCCCCTCCAGTGAGT
TCACCTCTAAATCACCACGATCAAGGAACAAGCATCAAGCACGCAATATGCAGCTC
AAAACGCTTAGCCTAGCCACACCTCACGGGAAACAGCAGTGATTAACCTTAGCAATAA
ACGAAAGTTTAACTAAGCTATACTACCCAGGGTTGGTCAATTTCTGTCACGCCACCGC
GGTCACACGATTAACCAAGTCAATCAAGCCGGCGTAAGAGTGTCTAGATCACCCCC
TCCCCAATAAAGCTAAACTCACTGATTTGTAAAAAACTCCAGTCTCAAAAAATAGAC
TACGAAAGTGGCTTTAACATATCTGAACCACTAAGCTAAGCTTGGGATTAGA
TACCCCACTATGCTTAGCCCTAACCTCAACAGCTTCAACAACTTGGCAGAA
CACTACGAGCCACAGCTTAAAACTCAAAGGACCTGGCGGTGCTTCATTAAGAGG
AGCCTGTTCTGTAATCGATAAAACCCGATCAACCTCACCACCTCTTGCTTCTAATA
CCGCCATCTTCAGCAACCCCTGATGAAGGCTACAAGTAAGCGCAAGTACCTGAGG
ACGTTAGGTCAAGGTGTAGCCCATGAGGTGGCAAGAAATGGGCTACATTTTCT
AAAACACGATAGCCCTTATGAACCTTAAGGTCGAAGGTGGATTTAGCAGTAA
AGTAGAGTGCTTAGTTGAACAGGGCCCTGAAGCGCTACACACCGCCGTCACCTT
AAGTATACTTCAAAGGACATTTAACTAAAACCCCTACGCATTTATATAGAGGAGACAA
CGTAACCTCAAACCTCTGCCTTTGGTGATCCACCCGCCCTTGGCCTACCTGCATAATGAAG
AAGCACCAACTTACCTTAGGAGATTTCAACTTAACTTGACCGCTCTGAGCTAACCTA
GCCCCAAACCCACTCCACCTTACTACAGACAACCTTAGCCAAACCATTTACCCAAATAA
AGTATAGGCGATAGAAATTGAAACCTGGCGCAATAGATATAGTACCGCAAGGAAAGATG
AAAAATTATAACCAAGCATAATATAGCAAGGACTAACCCCTATACCTTCTGCATAATGAA
TTAACTAGAAATAACTTTGCAAGGAGAGCCAAAGCTAAGACCCCCGAAACAGACGAGCT
ACCTAAGAACAGCTAAAAGAGCACACCCGTCTATGTAGCAAAATAGTGGGAAGATTTATA
GGTAGAGGCGACAAACCTACCGAGCCTGGTGATAGCTGGTTGTCCAAGATAGAATCTTAG
TTCAACTTTAAATTTGCCCCACAGAACCCTCTAAATCCCCTTGTAATTTAACTGTTAGTC
CAAAGAGGAACAGCTCTTTGGACACTAGGAAAAACCTTGTAGAGAGAGTAAAAATTTA
ACACCCATAGTAGGCCATAAAGCAGCCACCAATTAAGAAAGCGTTCAAGCTCAACACCA
CTACCTAAAAATCCCAAACATATAACTGAACTCCTCACACCAATTGGACCAATCTATC
ACCTATAGAAGAACTAATGTTAGTATAAGTAACATGAAAACATTCTCTCCGCATAAGC
CTGCGTCAGATTAAACACTGAACTGACAATTAACAGCCCAATATCTACAATCAACCAAC
AAGTCATTATTACCCTCACTGTCAACCCACACAGGCATGCTCATAAGGAAAGGTTAAAA
AAAGTAAAAGGAACTCGGCAAATCTTACCCCGCCTGTTTACCAAAAACATCACCTCTAGC
ATCACCAGTATTAGAGGCACCGCCTGCCAGTGACACATGTTTAAACGGCCGCGGTACCT
AACCGTGCAAAGGTAGCATAATCACTTGTTCCTTAAATAGGGACCTGTATGAATGGCTCC

Sequence differences occur because of...

1. Sequencing error
2. Genetic variation

Approximate matching

T: GGAAAAAGAGGTAGCGGCGTTTAACAGTAG

||| |||||

P: GTAACGGCG



Mismatch (Substitution)

Approximate matching

T: GGAAAAGAGGTAGC-GCGTTTAACAGTAG

||||| |||

P: GTAGCGGCG



Insertion

Approximate matching

T: GGAAAAAGAGGTAGCGGCGTTTAACAGTAG
 || |||||
P: GT-GCGGCG
 ↑
 Deletion

Hamming distance

For X & Y where $|X| = |Y|$, *hamming distance* =
minimum # substitutions needed to turn one into the other

X:	G	A	G	G	T	A	G	C	G	G	C	G	T	T
Y:	G	T	G	G	T	A	A	C	G	G	G	G	T	T

Hamming distance = 3

Edit distance

(AKA Levenshtein distance)

For X & Y , *edit distance* = minimum # edits (substitutions, insertions, deletions) needed 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

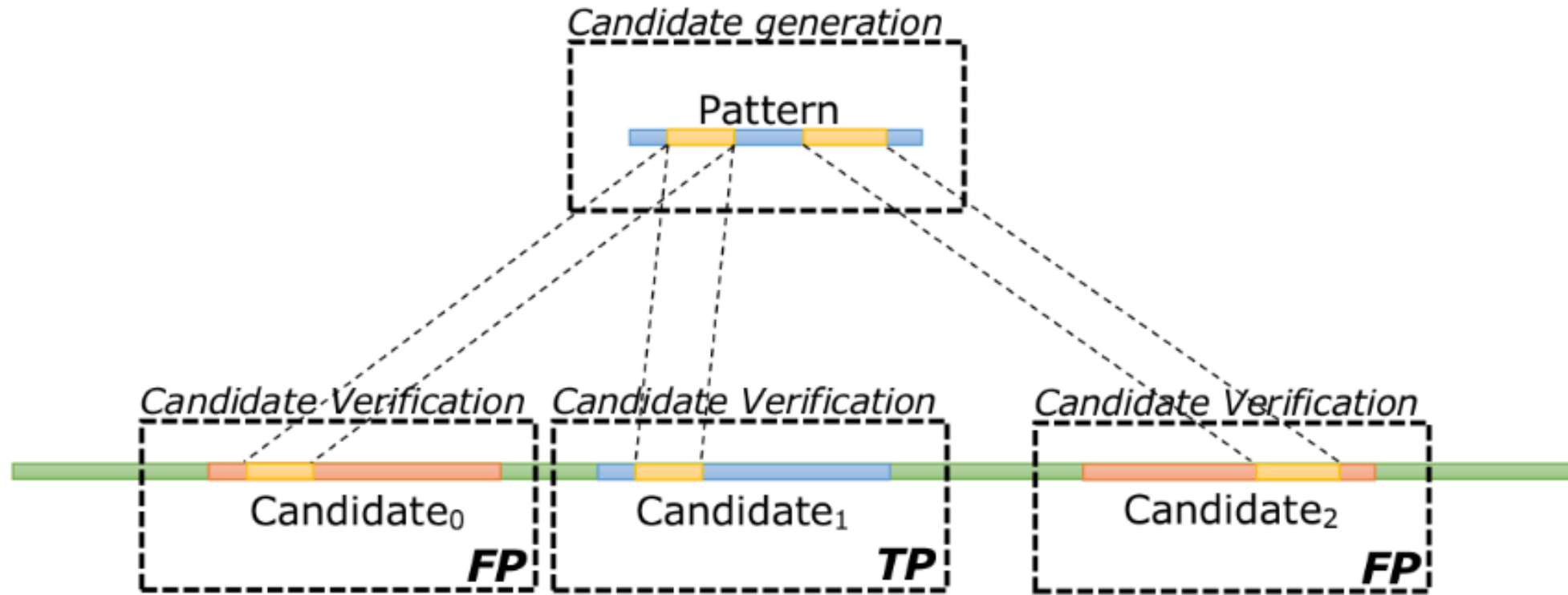
Edit distance = 2

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



Hamming sequence search in reference

- Implement a program that, given a reference sequence, searches a match for an input sequence up to m mismatches using Hamming distance.

Example

```
> \time -v python3 search_hamming.py -r ../../data/chr1.fa -i GAGCAATAAATT -m 2
```

```
Reading input FASTA ... read 248956422 bases from '../../data/chr1.fa'.  
Searching 'GAGCAATAAATT' sequence ...  
Found 9167 exact matches in 242.365 s.
```

Exercises

Hamming sequence search in reference

- Implement a program that, given a reference sequence, searches a match for an input sequence up to m mismatches using Hamming distance.

Code

```
def hamming_distance(seq1, seq2):  
    """Compute the Hamming distance between two sequences of equal length."""  
    if len(seq1) != len(seq2):  
        raise ValueError("Sequences must be of the same length")  
    distance = 0  
    for i in range(len(seq1)):  
        if seq1[i] != seq2[i]:  
            distance += 1  
    return distance
```

Hamming sequence search in reference

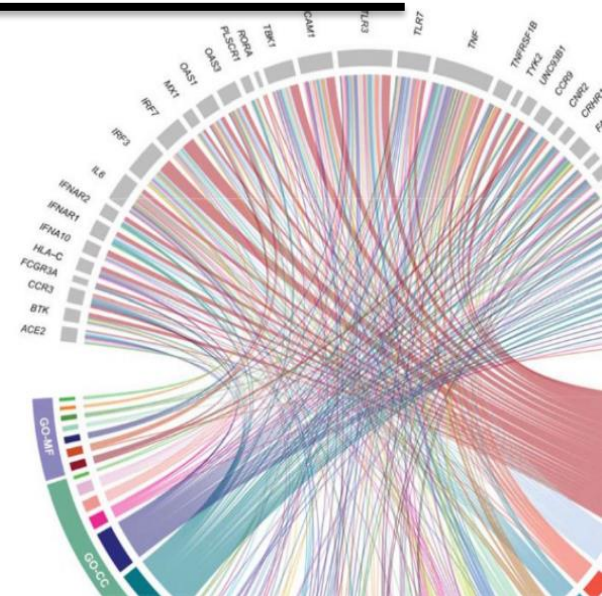
- Implement a program that, given a reference sequence, searches a match for an input sequence up to m mismatches using Hamming distance.

Code

```
def search_sequence(reference, sequence, m):
    start_time = time.time()
    matches = []
    for i in range(0, len(reference) - len(sequence)):
        if hamming_distance(reference[i:i + len(sequence)], sequence) <= m:
            matches.append(i)
    end_time = time.time()
    # Print Results
    print("Found %d exact matches in %2.3f s." % (len(matches), end_time - start_time))
    for match in matches:
        print("%d " % match, end="")
```

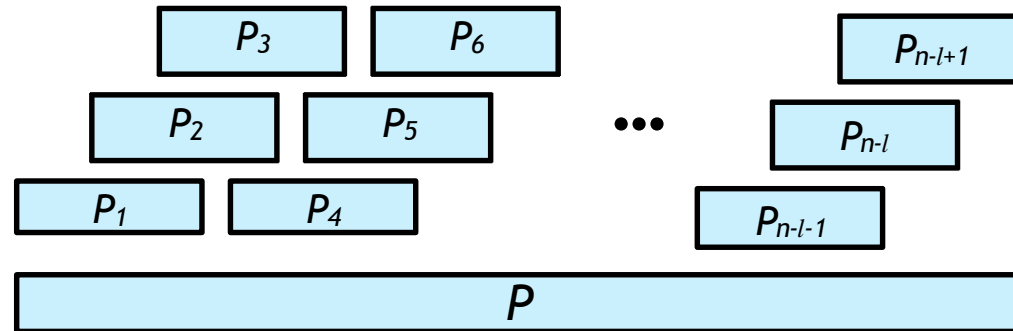
2

K-mer Seeding

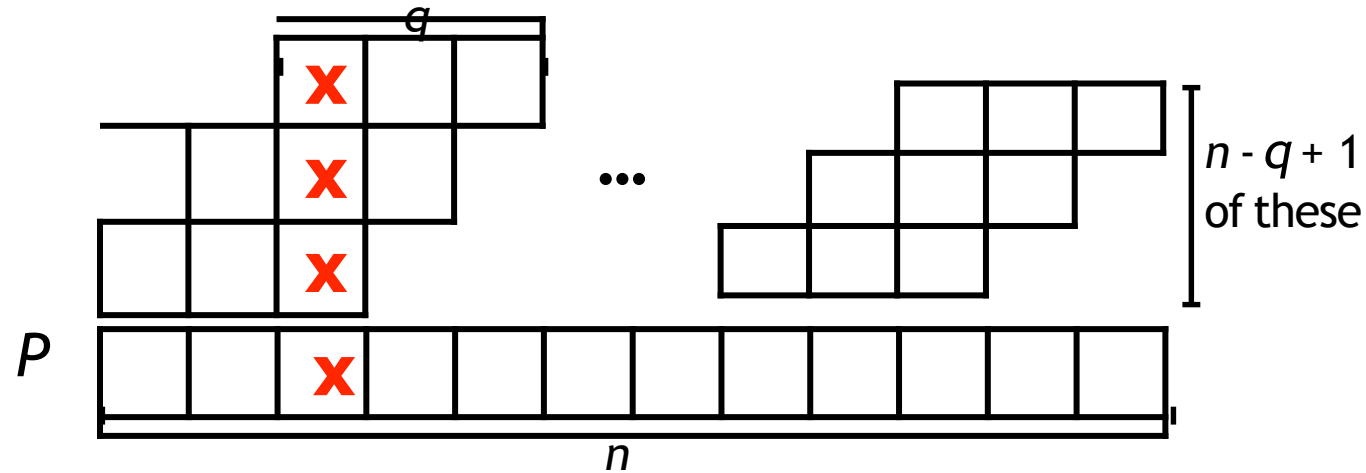


Overlapping K-mers

Now consider *overlapping* substrings



Q-gram Lemma



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

1 edit to P changes *at most* q substrings

*kq is worst case;
could be $< kq$*

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 using Q-gram Lemma

- If P occurs in T with up to k edits, alignment contains an exact match of length q , where $q \geq \text{floor}(n / (k + 1))$
 - Obtained by solving for q : $n - q + 1 - kq \geq 1$
- Exact matching filter
 - Find matches of length $\text{floor}(n / (k + 1))$ between T and any substring of P . Check vicinity for full match.

ACACCAACACG**T**GAGAC**T**GACTGACG

ACACCA CACG**T**G GAC**T**GA

CACCAA ACG**T**GA AC**T**GAC

ACCAAC CG**T**GAG CT**T**GACT

CCAACA GT**T**GAGA **T**GACTG

CAACAC **T**GAGAC GACTGA

AACACG GAGAC**T** ACTGAC

ACACG**T** AGACT**T**G CTGACG

...ACACCAACACG**C**GAGAC**A**GACTGACG...

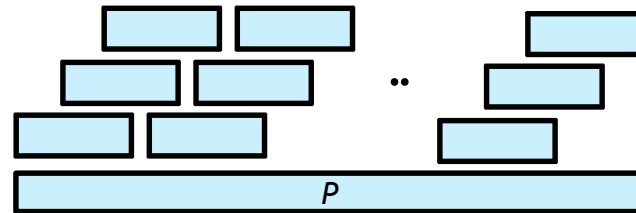
Lemma 5.3.2 (q-gram lemma). Given P , e , and $d()$, if the candidate w aligns the pattern P (i.e. $d(w, P) \leq e$) then P and w share at least $|w| + 1 - |q| \times (e + 1)$ common q-grams.

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 usually much speedier & still acceptably sensitive.



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

1. Sequence comparison by counting k-mers

- Given 2 sequences, m (maximum number of mismatches), and q (kmer length) use the q -gram lemma to compare the sequences and assess if they could match.

Code

```
from collections import Counter

def q_gram_match(T, P, q, m):
    """Uses the q-gram Lemma to assess if P occurs in T with at most k edits."""
    n = len(P)
    num_qgrams_needed = max(1, n - q + 1 - m * q) # Minimum q-gram matches required
    # Generate q-grams for P
    P_qgrams = [P[i:i+q] for i in range(len(P) - q + 1)]
    P_qgram_counts = Counter(P_qgrams)
    # Generate q-grams for T
    T_qgrams = [T[i:i+q] for i in range(len(T) - q + 1)]
    T_qgram_counts = Counter(T_qgrams)
    # Count how many q-grams in P appear in T
    matching_qgrams = \
        sum(min(P_qgram_counts[qgram], T_qgram_counts.get(qgram, 0)) for qgram in P_qgram_counts)
    # Decision based on q-gram lemma
    return matching_qgrams >= num_qgrams_needed
```

Exercises

1. Sequence comparison by counting k-mers

- Given 2 sequences, m (maximum number of mismatches), and q (kmer length) use the q -gram lemma to compare the sequences and assess if they could match.

Code

```
# Example Usage:
T = "GATCACAGGTCTATCACCTATTAACCACT"
#           ||| ||||| ||
P = "GGTGTATCACTCT"
q = 3
m = 1

result = q_gram_match(T, P, q, m)
print(f"Does P appear in T with  $\leq \{m\}$  edits? {result}")
```

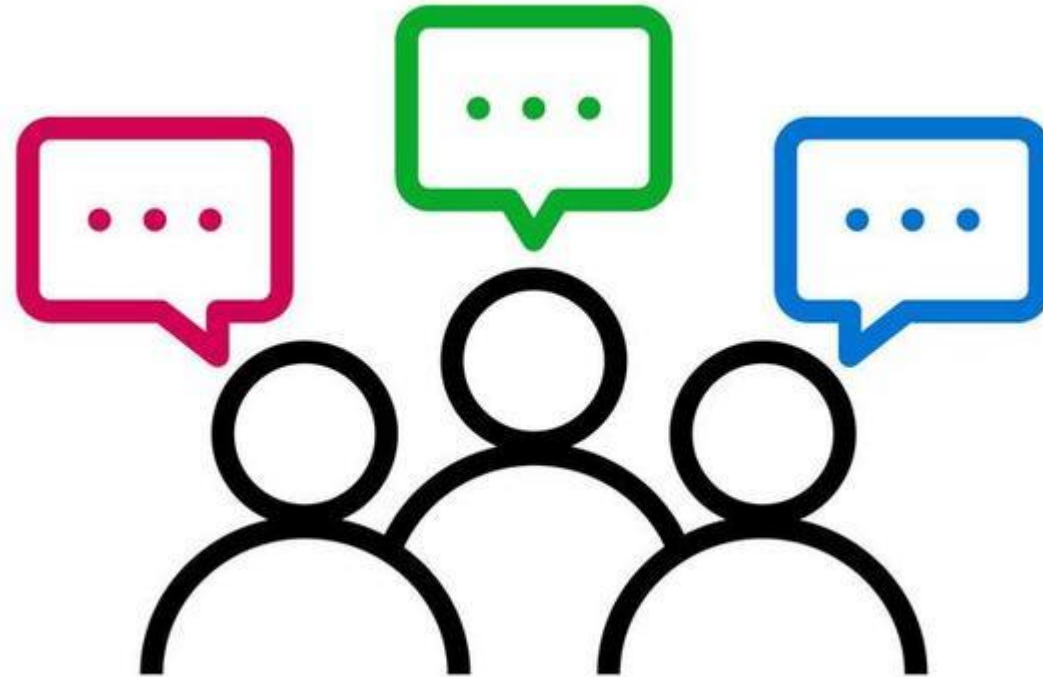
2. Online Approximate String Matching (using q-gram lemma)

- Improve the online search program to exploit the q-gram lemma and find all matches in the reference (chr1.fa) that match a given input-sequence with less than m mismatches.



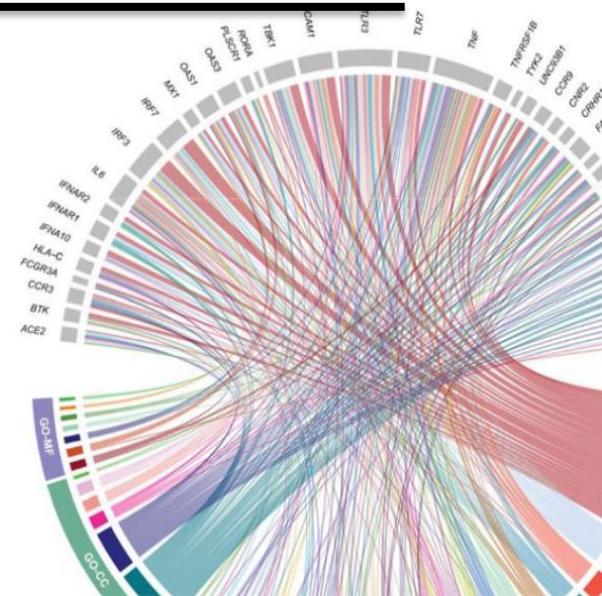
3. Indexed Approximate String Matching (using q-gram lemma)

- Improve the kmer-hash based program to exploit the q-gram lemma and find all matches in the reference (chr1.fa) that match a given input-sequence with less than m mismatches.

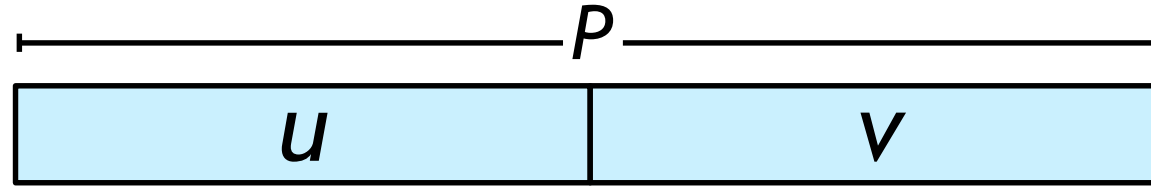


3

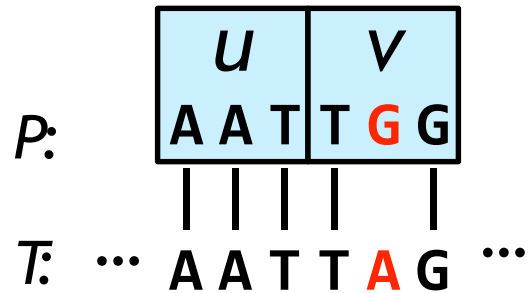
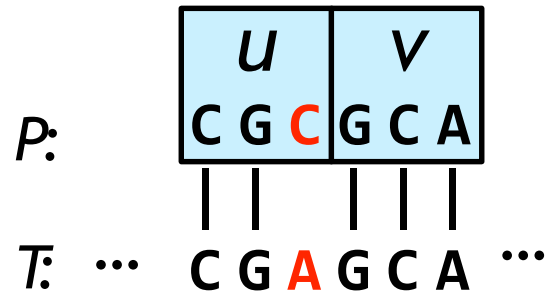
Factors Seeding



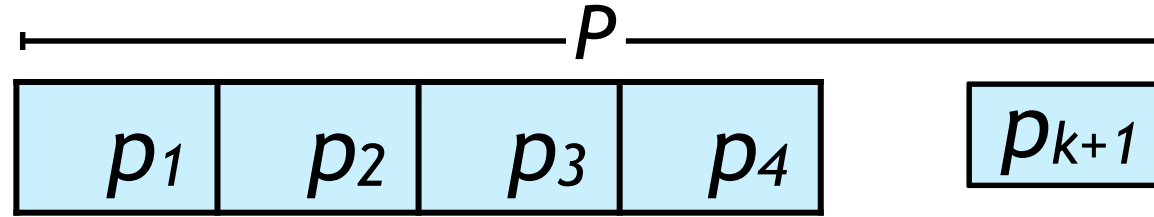
Approximate matching



If P occurs in T with 1 edit, then u or v appears with no edits

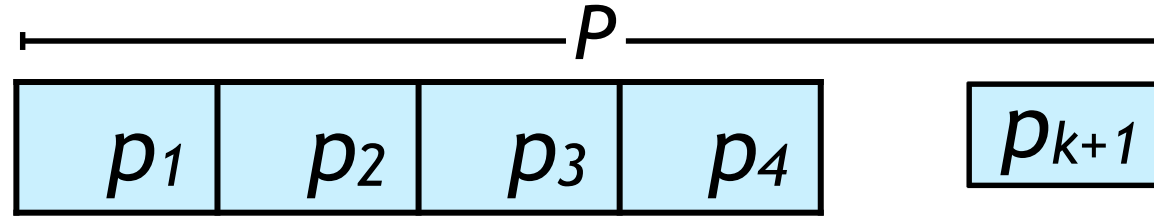


Approximate matching



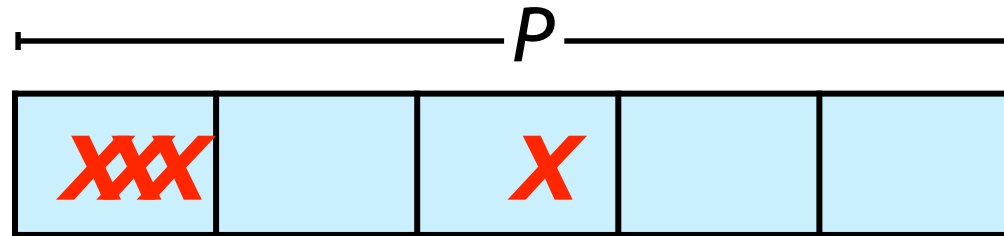
If P occurs in T with up to k edits...

Approximate matching

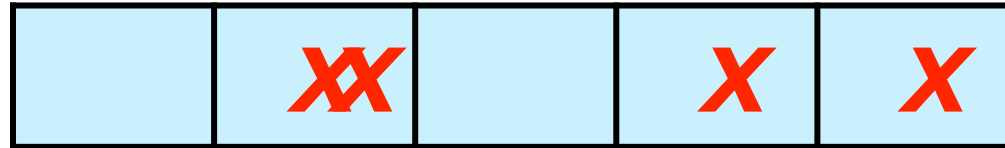


If P occurs in T with up to k edits, then at least one of p_1, p_2, \dots, p_{k+1} must appear with 0 edits

Approximate matching



5 partitions
4 edits (~~X~~)



Approximate matching



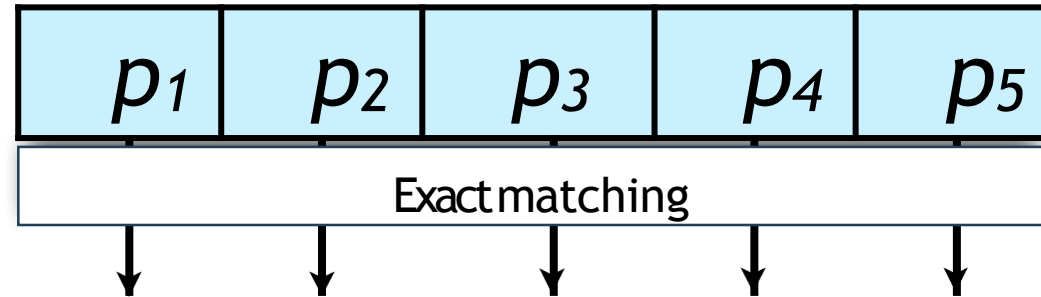
Pigeonhole principle: $k+1$ pigeons, k holes.
At least one has > 1 pigeon!

Approximate matching



We have k pigeons, $k+1$ holes, at least one...
...is *empty*

Pigeonhole Principle



What algorithm can we use?

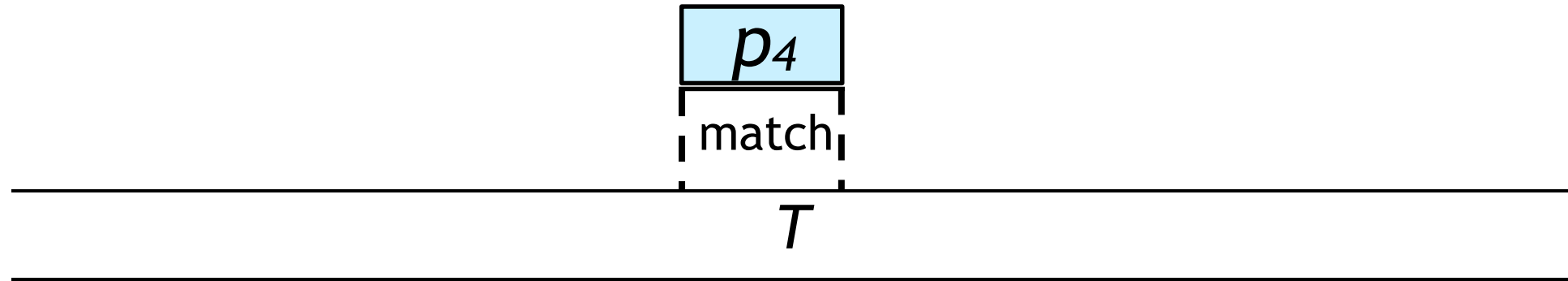
Any exact matching algorithm

If we have a k-mer index, we can use that

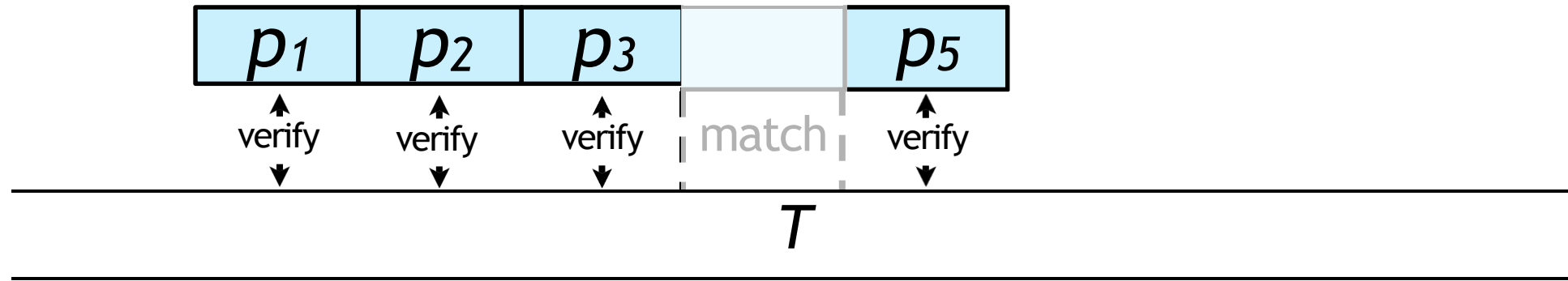
Naive exact matching

Boyer-Moore

Pigeonhole Principle

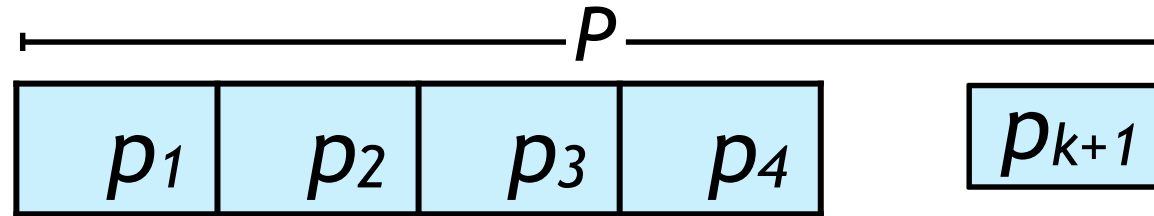


Pigeonhole Principle



For Hamming distance, verification is essentially just the inner loop of **naive_approx_hamming** from before

Pigeonhole Principle



Advantages

Reuse favorite exact matching algos; fast and easy

Flexible; works for Hamming and edit distance*

Disadvantages

Large k yields small partitions matching many times by chance; lots of verification work

$k+1$ exact matching problems, one per partition

* we don't know how to do edit distance verification yet

Pigeonhole Principle

Implementation of pigeonhole principle with Boyer-Moore as exact matching algorithm:

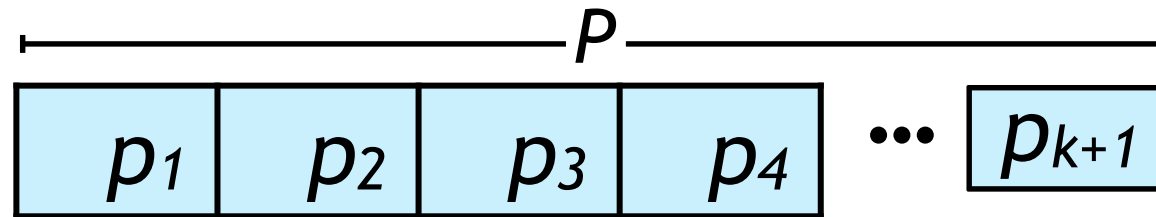
http://j.mp/CG_ApproxBM

	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	1.91 s	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

Generalizing pigeonhole, part 1

If P occurs in T with up to k edits, then at least one of p_1, p_2, \dots, p_{k+1} must appear with 0 edits



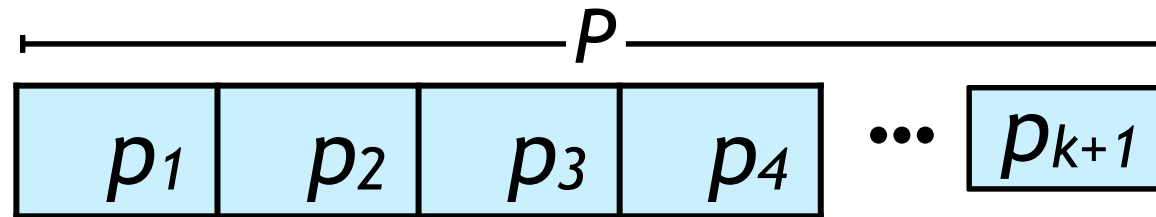
But doesn't *have to* be “at least one of” ...

what would we have to change for “at least two of”?

If P occurs in T with up to k edits, then at least **two** of _____ must appear with 0 edits

Generalizing pigeonhole, part 1

If P occurs in T with up to k edits, then at least one of p_1, p_2, \dots, p_{k+1} must appear with 0 edits



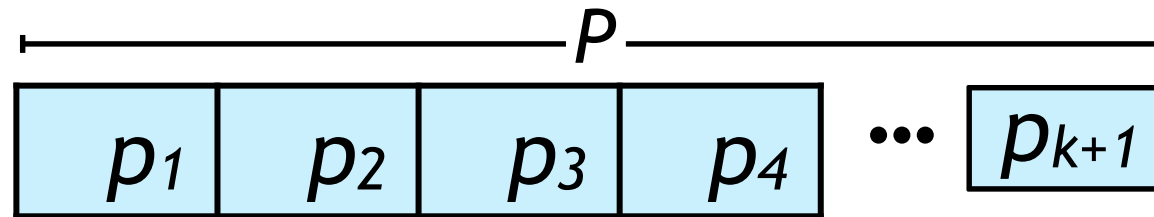
But doesn't *have to* be “at least one of” ...

what would we have to change for “at least two of”?

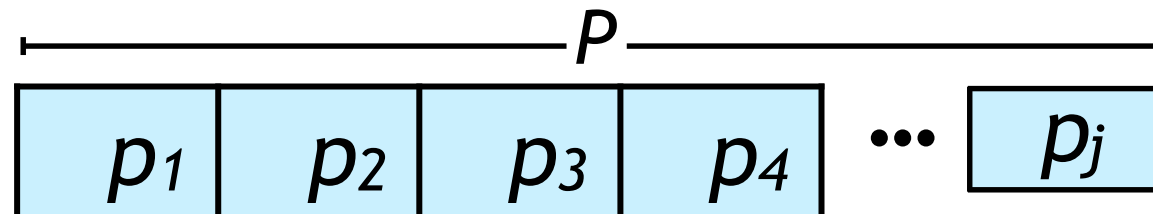
If P occurs in T with up to k edits, then at least **two** of p_1, p_2, \dots, p_{k+2} must appear with 0 edits

Generalizing pigeonhole, part 2

If P occurs in T with up to k edits, then at least one of p_1, p_2, \dots, p_{k+1} must appear with 0 edits

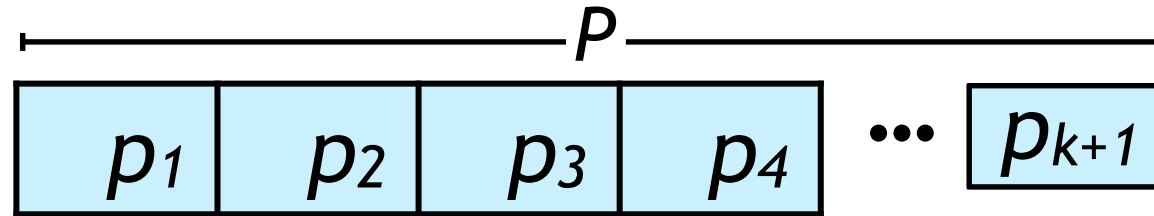


Let p_1, p_2, \dots, p_j be a partitioning of P . If P occurs with up to k edits, then at least one of p_1, p_2, \dots, p_j must occur with \leq ??? edits.

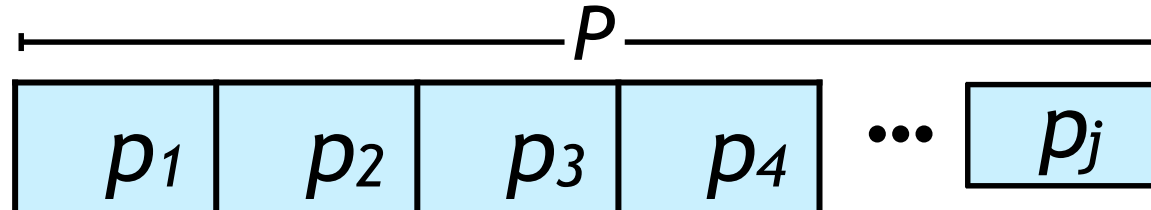


Generalizing pigeonhole, part 2

If P occurs in T with up to k edits, then at least one of p_1, p_2, \dots, p_{k+1} must appear with 0 edits



Let p_1, p_2, \dots, p_j be a partitioning of P . If P occurs with up to k edits, then at least one of p_1, p_2, \dots, p_j must occur with $\leq \text{floor}(k / j)$ edits.



Generalizing pigeonhole, part 2

X

X

X

X

X

X

X

X

X

XX

XX

X

XX

XX

XX

XX

XX

XX

XX

etc

At least one of
 p_1, p_2, \dots, p_5
occurs with...

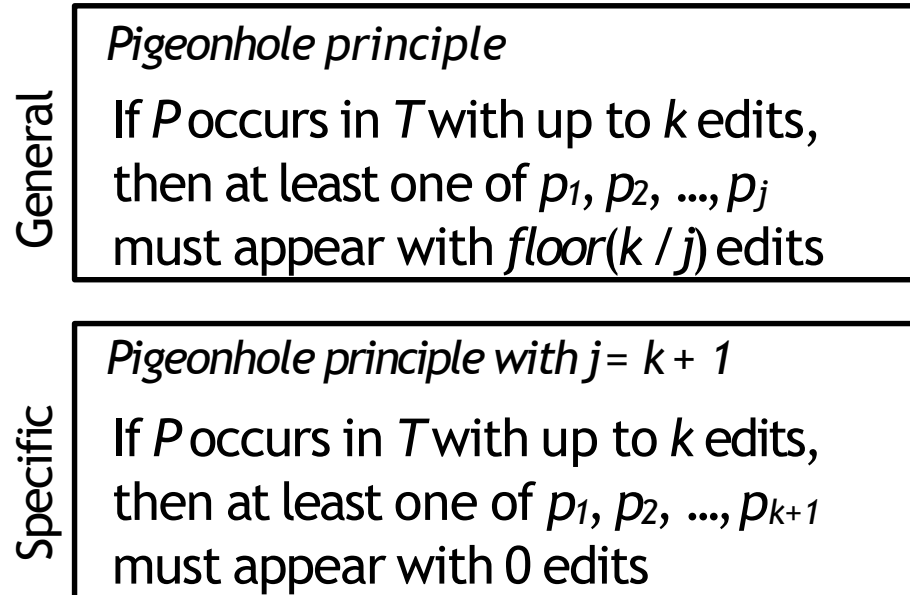
k = 4 edits ≤ 0 edits

k = 5 edits ≤ 1 edits

k = 9 edits ≤ 1 edits

k = 10 edits ≤ 2 edits

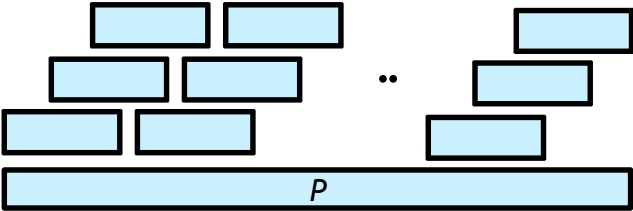
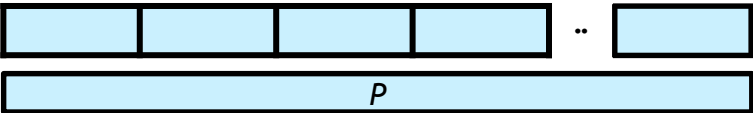
Generalizing pigeonhole, part 2



- Let $j = k + 1$
- *Why?*
- Smallest value s.t. $\text{floor}(k / j) = 0$
- *Why make $\text{floor}(k / j) = 0$?*
- So we can use exact matching
- *Why is smaller j good?*
- Yields fewer, longer partitions
- *Why are long partitions good?*
- Makes exact-matching filter more specific, minimizing # candidates

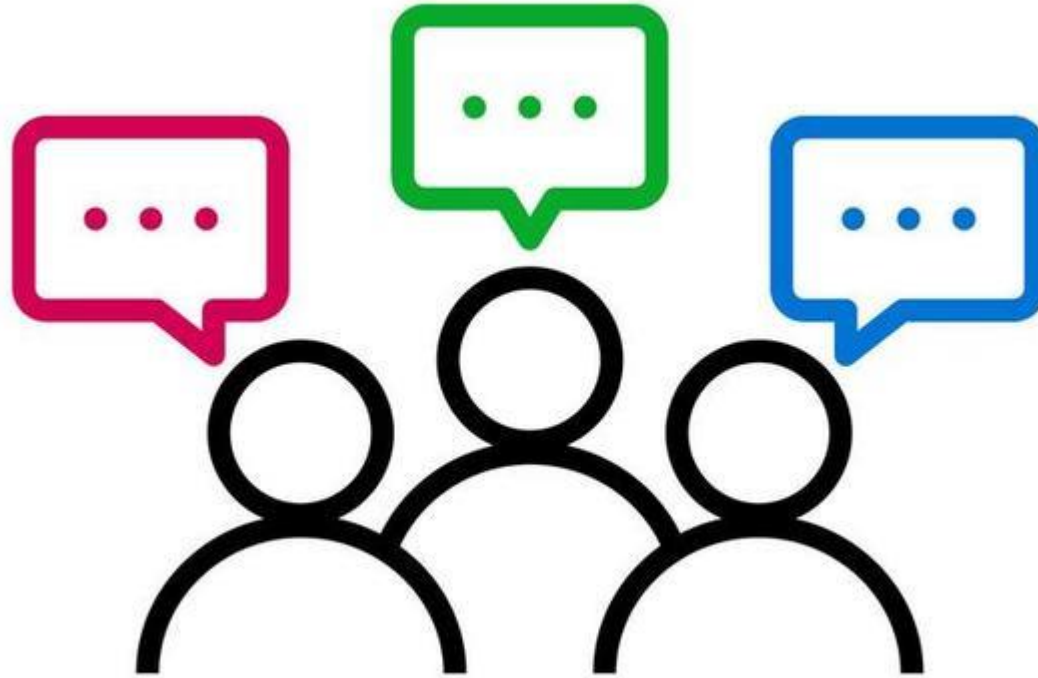
Approximate matching principles

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



1. Indexed Approximate String Matching (using factors)

- Improve the kmer-hash based program to exploit factor filters and find all matches in the reference (chr1.fa) that match a given input-sequence with less than m mismatches.



Questions

