Approximate matching

Ben Langmead

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Department of Computer Science

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:

```
T: GGAAAAAAGAGGTAGCGGCGTTTAACAGTAG
                |||   |||||||
P:              GTAACGGCG
```

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

```
T: GGAAAAAAGAGGTAGCGGCGTTTAACAGTAG
                |||   |||||||
P:              GTAACGGCG

                        Gap in T
                        ↙
T: GGAAAAAAGAGGTAGC-GCGTTTAACAGTAG
                |||||   |||||||
P:              GTAGCGGCG

T: GGAAAAAAGAGGTAGCGGCGTTTAACAGTAG
                ||   |||||||
P:              GT-GCGGCG

Gap in P
↗
```

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
```

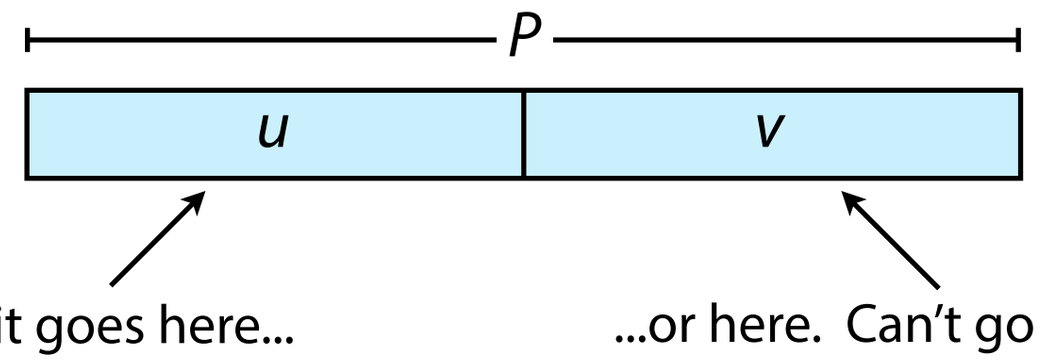
Instead of stopping upon first mismatch, stop when maximum distance is exceeded

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

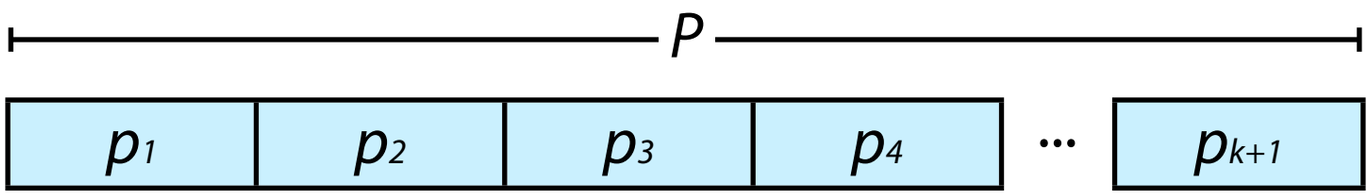
Approximate string matching

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* occurrs in *T* with 1 edit, either *u* or *v* must match exactly.



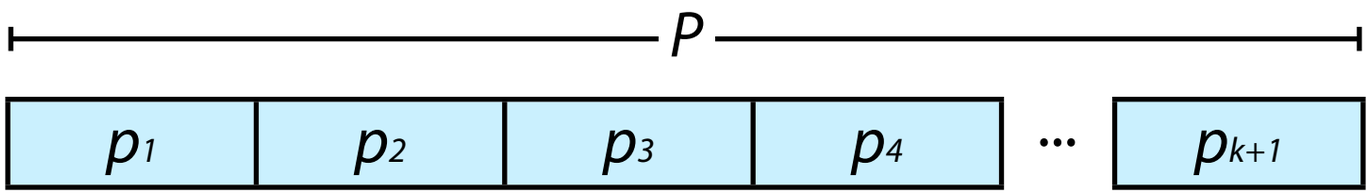
More generally: Let $p_1, p_2, ..., p_{k+1}$ be a partitioning of *P* into *k*+1 non-overlapping non-empty substrings. If *P* occurrs in *T* with up to *k* edits, then at least one of $p_1, p_2, ..., p_{k+1}$ must match exactly.



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

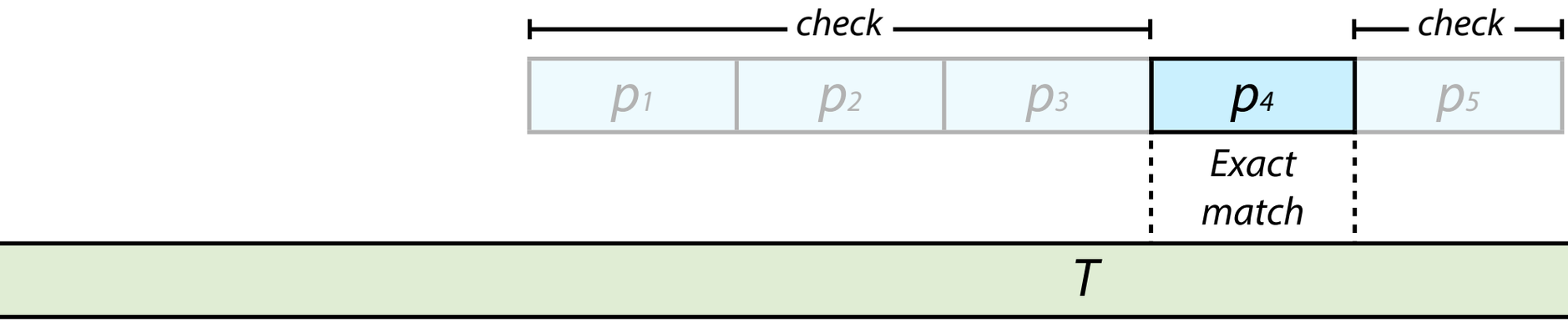
Approximate string matching

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, ..., 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

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

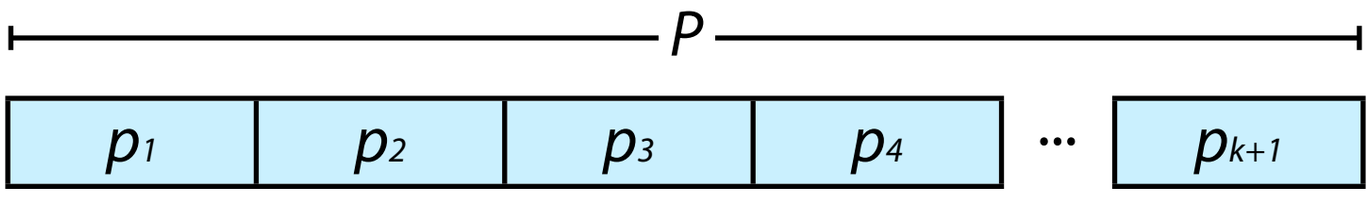
Approximate Boyer-Moore performance

	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

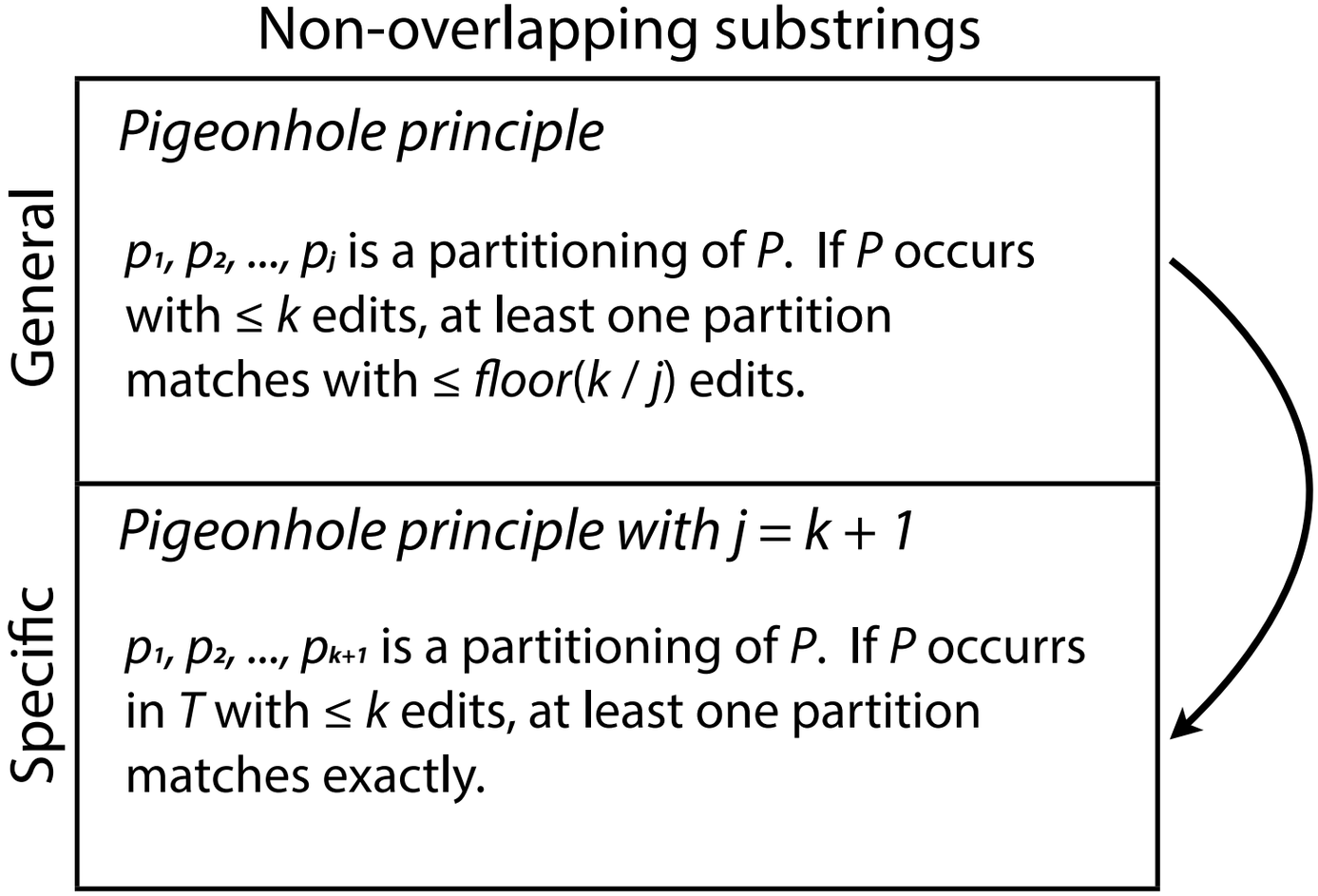
Let p_1, p_2, ..., p_{k+1} be a partitioning of P into k+1 non-overlapping non-empty substrings. If P occurrs in T with up to k edits, then at least one of p_1, p_2, ..., p_{k+1} must match exactly.



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 ≤ *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

Why is smaller j good?

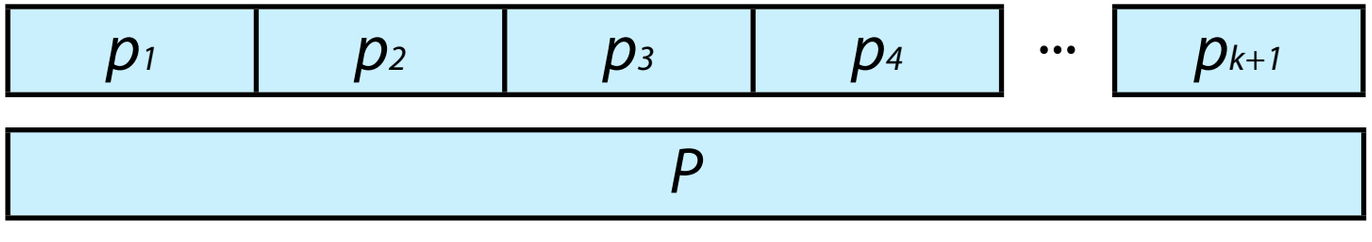
Yields fewer, longer partitions

Why are long partitions good?

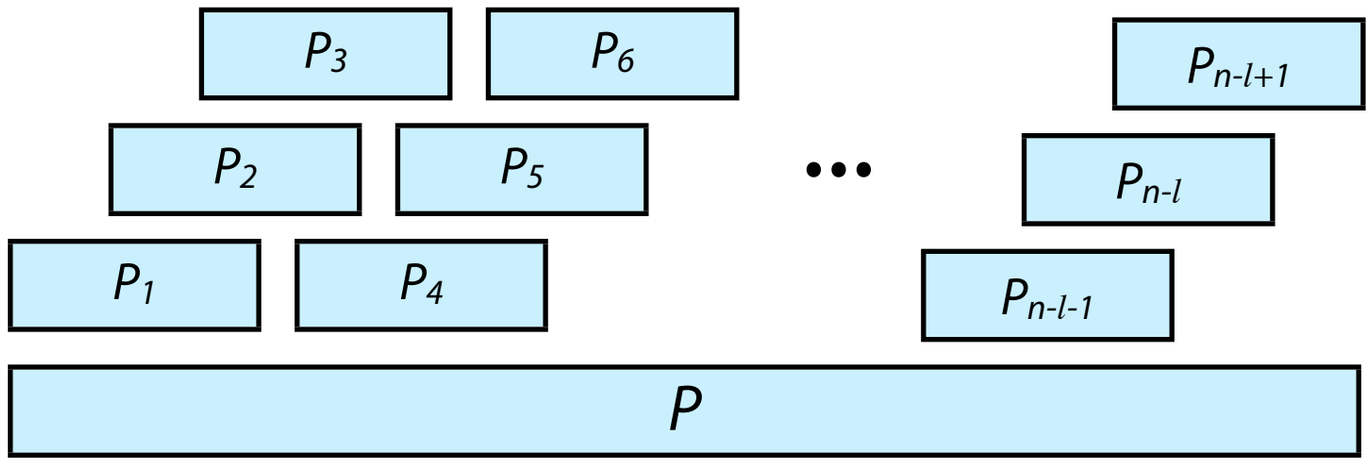
Makes exact-matching filter more specific, minimizing # candidates

Approximate string matching: more principles

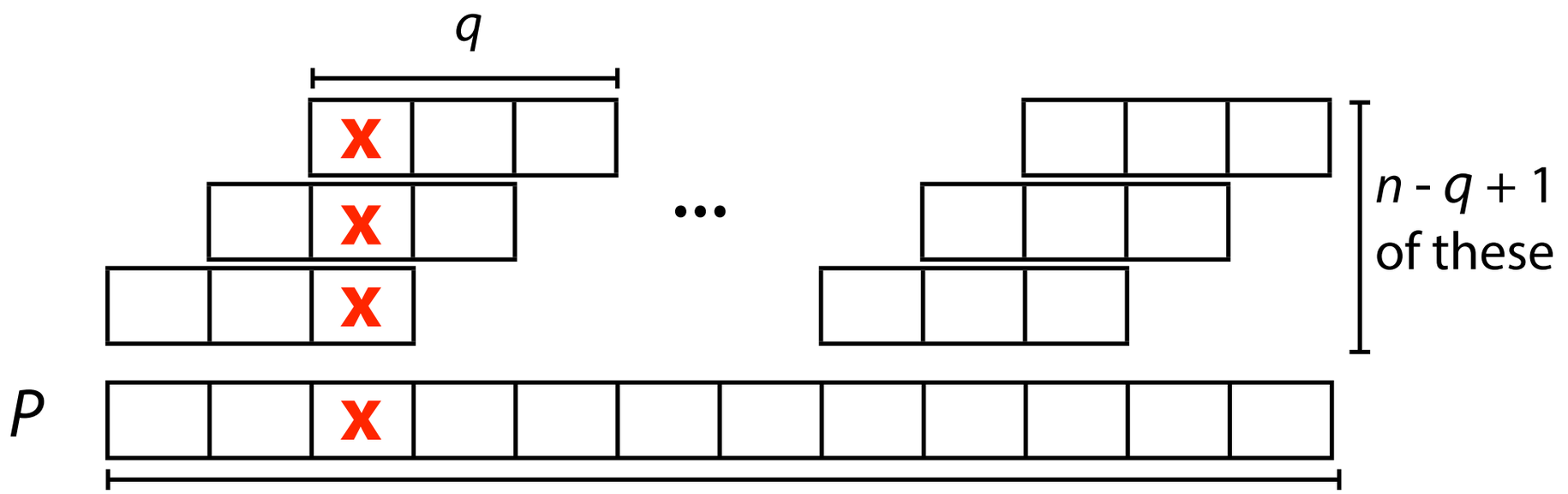
We partitioned *P* into non-overlapping substrings



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

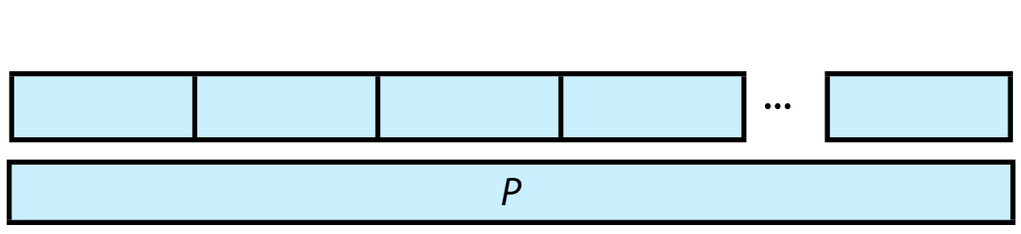
If *P* occurs in *T* with up to *k* edits, alignment contains an exact match of length *q,* where $q \geq floor(n / (k + 1))$

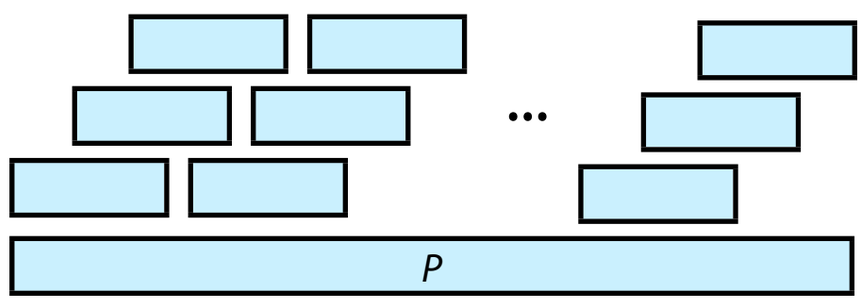
Derived by solving this for *q*: $n - q + 1 - kq \geq 1$

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



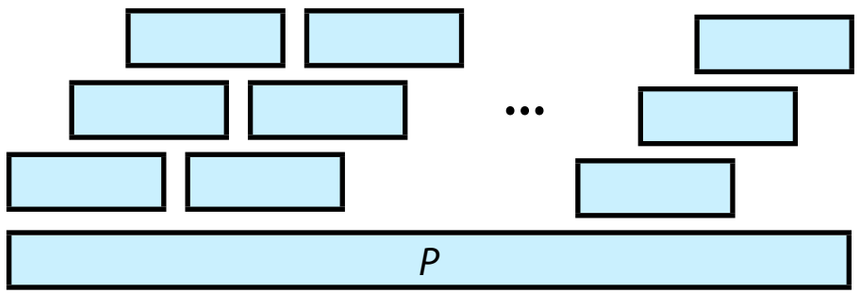


Sensitivity

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 $q > floor(n / (k + 1))$