This entire lecture is adopted (with permission) directly from a *legend* of the read-mapping world

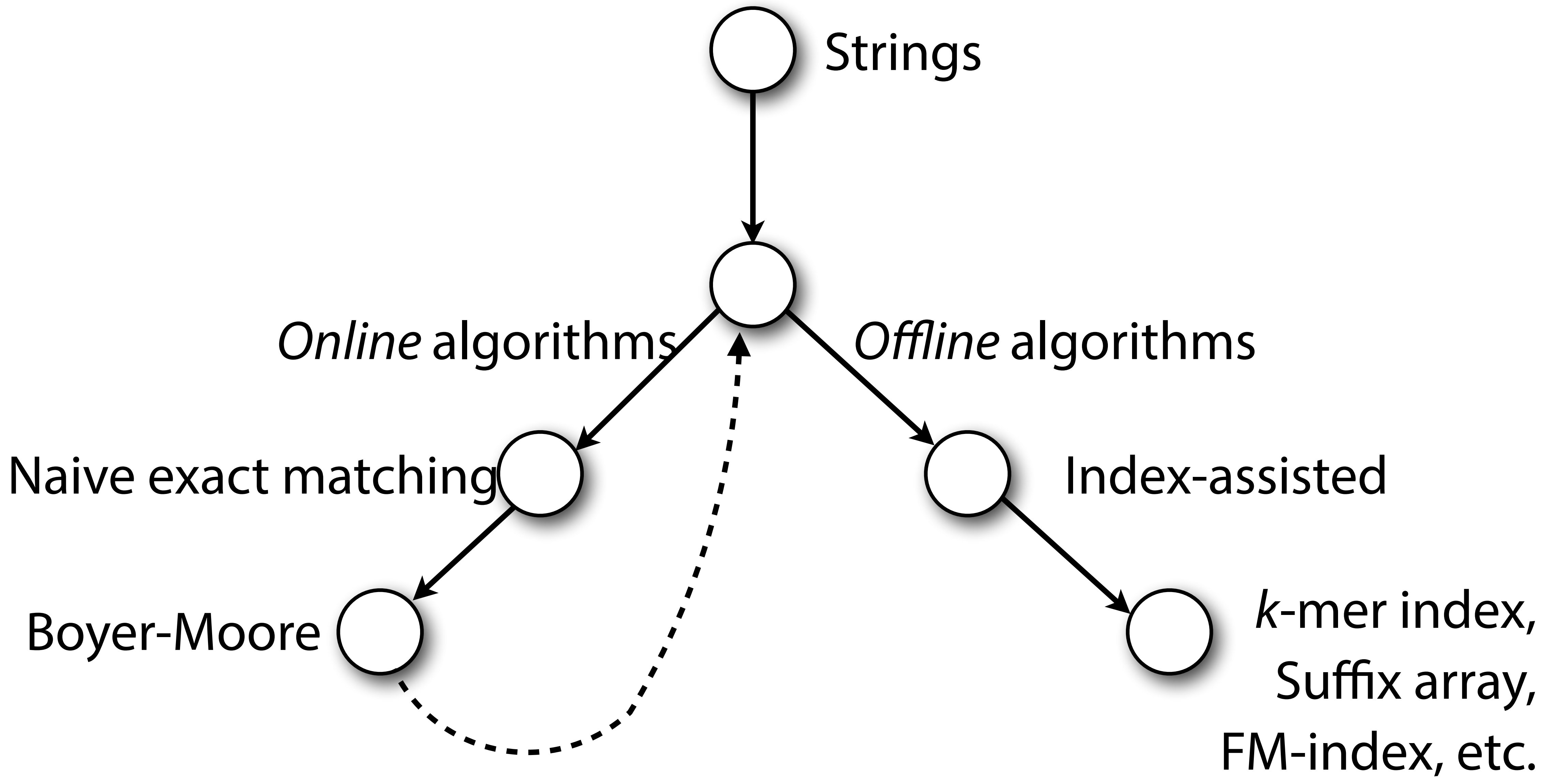
Approximate Matching

Ben Langmead

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Department of Computer Science

Approximate matching



We have focused on *exact* matching...

... in reality, we have to deal with *differences*

Read

```
CTCAAACTCCTGACCTTTGGTGATCCACCCGCCTNGGCCTTC
```

Reference

```
GATCACAGGTCTATCACCCTATTAACCACTCACGGGAGCTCTCCATGCATTTGGTATTTT
CGTCTGGGGGGTATGCACGCGATAGCATTGCGAGACGCTGGAGCCGGAGCACCCTATGTC
GCAGTATCTGTCTTTGATTCCTGCCTCATCCTATTATTTATCGCACCTACGTTCAATATT
ACAGGCGAACATACTTACTAAAGTGTGTTAATTAATTAATGCTTGTAGGACATAATAATA
ACAATTGAATGTCTGCACAGCCACTTTCCACACAGACATCATAACAAAAAATTTCCACCA
AACCCCCCCTCCCCCGCTTCTGGCCACAGC[illegible]AAACACA[illegible]CTCTGCCAAACCCCAAAA
ACAAAGAACCCTAACACCAGCCTAACCA[illegible]ATTTCAAATTTTAT[illegible]TTTGGCGGTATGCAC
TTTTAACAGTCACCCCCCAACTAACA[illegible]ATTATTTTCCCCTCCCAC[illegible]CATACTACTAAT
CTCATCAATACAACCCCCGCCCAT[illegible]TACCCAGCACACACACACCG[illegible]CTAACCCCATA
CCCCGAACCAACCAAACCCCAAAG[illegible]CACCCCCCACAGTTTATGTAGC[illegible]ACCTCCTCAAA
GCAATACACTGACCCGCTCAAAC[illegible]CCTGGATTTTGGATCCACCCAGCG[illegible]TTGGCCTAAA
CTAGCCTTTCTATTAGCTCTTAG[illegible]AAGATTACACATGCAAGCATCCCCG[illegible]TCCAGTGAGT
TCACCCTCTAAATCACCACGATC[illegible]AAAGGAACAAGCATCAAGCACGCAG[illegible]AATGCAGCTC
AAAACGCTTAGCCTAGCCACACC[illegible]CCACGGGAAACAGCAGTGATTAAC[illegible]TTAGCAATAA
ACGAAAGTTTAACTAAGCTATACT[illegible]ACCCCAGGGTTGGTCAATTTCGT[illegible]CCAGCCACCGC
GGTCACACGATTAACCCAAGTCAAT[illegible]GAAGCCGGCGTAAAGAGTGTT[illegible]TAGATCACCCCC
TCCCCAATAAAGCTAAAACTCACCTGA[illegible]TTGTAAAAAACTCCAGT[illegible]ACACAAAATAGAC
TACGAAAGTGGCTTTAACATATCTGAACA[illegible]CAATAGCTAAGA[illegible]GGGATTAGA
TACCCCACTATGCTTAGCCCTAAACCTCAACAG[illegible]AACAA[illegible]GCCAGAA
CACTACGAGCCACAGCTTAAAACTCAAAGGACCTGGCGGTGCTTCAT[illegible]TAGAGG
AGCCTGTTCTGTAATCGATAAACCCCGATCAACCTCACCACCTCTTGCT[illegible]TATA
CCGCCATCTTCAGCAAACCCTGATGAAGGCTACAAAGTAAGCGCAAGTACC[illegible]AG
ACGTTAGGTCAAGGTGTAGCCCATGAGGTGGCAAGAAATGGGCTACATTTTCT[illegible]
AAAACTACGATAGCCCTTATGAAACTTAAGGGTCGAAGGTGGATTTAGCAGTAAA[illegible]
AGTAGAGTGCTTAGTTGAACAGGGCCCTGAAGCGCGTACACACCGCCCGTCACCCT[illegible]
AAGTATACTTCAAAGGACATTTAACTAAAACCCCTACGCATTTATATAGAGGAGACAA[illegible]
CGTAACCTCAAACTCCTGCCTTTGGTGATCCACCCGCCTTGGCCTACCTGCATAATGAAG
AAGCACCCAACTTACACTTAGGAGATTTCAACTTAACTTGACCGCTCTGAGCTAAACCTA
GCCCCAAACCCACTCCACCTTACTACCAGACAACCTTAGCCAAACCATTTACCCAAATAA
AGTATAGGCGATAGAAATTGAAACCTGGCGCAATAGATATAGTACCGCAAGGGAAAGATG
AAAAATTATAACCAAGCATAATATAGCAAGGACTAACCCCTATACCTTCTGCATAATGAA
TTAACTAGAAATAACTTTGCAAGGAGAGCCAAAGCTAAGACCCCCGAAACCAGACGAGCT
ACCTAAGAACAGCTAAAAGAGCACACCCGTCTATGTAGCAAAATAGTGGGAAGATTTATA
GGTAGAGGCGACAAACCTACCGAGCCTGGTGATAGCTGGTTGTCCAAGATAGAATCTTAG
TTCAACTTTAAATTTGCCCACAGAACCCTCTAAATCCCCTTGTAAATTTAACTGTTAGTC
CAAAGAGGAACAGCTCTTTGGACACTAGGAAAAAACCTTGTAGAGAGAGTAAAAAATTTA
ACACCCATAGTAGGCCTAAAAGCAGCCACCAATTAAGAAAGCGTTCAAGCTCAACACCCA
CTACCTAAAAAATCCCAAACATATAACTGAACTCCTCACACCCAATTGGACCAATCTATC
ACCCTATAGAAGAACTAATGTTAGTATAAGTAACATGAAAACATTCTCCTCCGCATAAGC
CTGCGTCAGATTAAAACACTGAACTGACAATTAACAGCCCAATATCTACAATCAACCAAC
AAGTCATTATTACCCTCACTGTCAACCCAACACAGGCATGCTCATAAGGAAAGGTTAAAA
AAAGTAAAAGGAACTCGGCAAATCTTACCCCGCCTGTTTACCAAAAACATCACCTCTAGC
ATCACCAGTATTAGAGGCACCGCCTGCCCAGTGACACATGTTTAACGGCCGCGGTACCCT
AACCGTGCAAAGGTAGCATAATCACTTGTTCCTTAAATAGGGACCTGTATGAATGGCTCC
```

Sequence differences occur because of...

Read

```
CTCAAACTCCTGACCTTTGGTGATCCACCCGCCTNGGCCTTC
```

Reference

```
GATCACAGGTCTATCACCCTATTAACCACTCACGGGAGCTCTCCATGCATTTGGTATTTT
CGTCTGGGGGGTATGCACGCGATAGCATTGCGAGACGCTGGAGCCGGAGCACCCTATGTC
GCAGTATCTGTCTTTGATTCCTGCCTCATCCTATTATTTATCGCACCTACGTTCAATATT
ACAGGCGAACATACTTACTAAAGTGTGTTAATTAATTAATGCTTGTAGGACATAATAATA
ACAATTGAATGTCTGCACAGCCACTTTCCACACAGACATCATAACAAAAAATTTCCACCA
AACCCCCCCTCCCCCGCTTCTGGCCACAGC[illegible]TAAACACA[illegible]TCTGCCAAACCCCAAAA
ACAAAGAACCCTAACACCAGCCTAACCA[illegible]ATTTCAAATTTTATC[illegible]TTTGGCGGTATGCAC
TTTTAACAGTCACCCCCCAACTAACA[illegible]ATTATTTTCCCCTCCCAC[illegible]CATACTACTAAT
CTCATCAATACAACCCCCGCCCATC[illegible]TACCCAGCACACACACACCG[illegible]CTAACCCCATA
CCCCGAACCAACCAAACCCCAAAG[illegible]CACCCCCCACAGTTTATGTAGC[illegible]ACCTCCTCAAA
GCAATACACTGACCCGCTCAAAC[illegible]CCTGGATTTTGGATCCACCCAGCG[illegible]TTGGCCTAAA
CTAGCCTTTCTATTAGCTCTTAG[illegible]AAGATTACACATGCAAGCATCCCCG[illegible]TCCAGTGAGT
TCACCCTCTAAATCACCACGATC[illegible]AAAGGAACAAGCATCAAGCACGCAG[illegible]AATGCAGCTC
AAAACGCTTAGCCTAGCCACACCC[illegible]CACGGGAAACAGCAGTGATTAAC[illegible]TTAGCAATAA
ACGAAAGTTTAACTAAGCTATACT[illegible]ACCCCAGGGTTGGTCAATTTCGT[illegible]CCAGCCACCGC
GGTCACACGATTAACCCAAGTCAAT[illegible]GAAGCCGGCGTAAAGAGTGTT[illegible]TAGATCACCCCC
TCCCCAATAAAGCTAAAACTCACCTGA[illegible]TTGTAAAAAACTCCAGT[illegible]ACACAAAATAGAC
TACGAAAGTGGCTTTAACATATCTGAACA[illegible]ACAATAGCTAAGA[illegible]TGGGATTAGA
TACCCCACTATGCTTAGCCCTAAACCTCAACAG[illegible]CAACAAA[illegible]CGCCAGAA
CACTACGAGCCACAGCTTAAAACTCAAAGGACCTGGCGGTGCTTCATA[illegible]TAGAGG
AGCCTGTTCTGTAATCGATAAACCCCGATCAACCTCACCACCTCTTGCT[illegible]TATA
CCGCCATCTTCAGCAAACCCTGATGAAGGCTACAAAGTAAGCGCAAGTACC[illegible]AG
ACGTTAGGTCAAGGTGTAGCCCATGAGGTGGCAAGAAATGGGCTACATTTTCT[illegible]
AAAACTACGATAGCCCTTATGAAACTTAAGGGTCGAAGGTGGATTTAGCAGTAAA[illegible]
AGTAGAGTGCTTAGTTGAACAGGGCCCTGAAGCGCGTACACACCGCCCGTCACCCT[illegible]
AAGTATACTTCAAAGGACATTTAACTAAAACCCCTACGCATTTATATAGAGGAGACAA[illegible]
CGTAACCTCAAACTCCTGCCTTTGGTGATCCACCCGCCTTGGCCTACCTGCATAATGAAG
AAGCACCCAACTTACACTTAGGAGATTTCAACTTAACTTGACCGCTCTGAGCTAAACCTA
GCCCCAAACCCACTCCACCTTACTACCAGACAACCTTAGCCAAACCATTTACCCAAATAA
AGTATAGGCGATAGAAATTGAAACCTGGCGCAATAGATATAGTACCGCAAGGGAAAGATG
AAAAATTATAACCAAGCATAATATAGCAAGGACTAACCCCTATACCTTCTGCATAATGAA
TTAACTAGAAATAACTTTGCAAGGAGAGCCAAAGCTAAGACCCCCGAAACCAGACGAGCT
ACCTAAGAACAGCTAAAAGAGCACACCCGTCTATGTAGCAAAATAGTGGGAAGATTTATA
GGTAGAGGCGACAAACCTACCGAGCCTGGTGATAGCTGGTTGTCCAAGATAGAATCTTAG
TTCAACTTTAAATTTGCCCACAGAACCCTCTAAATCCCCTTGTAAATTTAACTGTTAGTC
CAAAGAGGAACAGCTCTTTGGACACTAGGAAAAAACCTTGTAGAGAGAGTAAAAAATTTA
ACACCCATAGTAGGCCTAAAAGCAGCCACCAATTAAGAAAGCGTTCAAGCTCAACACCCA
CTACCTAAAAAATCCCAAACATATAACTGAACTCCTCACACCCAATTGGACCAATCTATC
ACCCTATAGAAGAACTAATGTTAGTATAAGTAACATGAAAACATTCTCCTCCGCATAAGC
CTGCGTCAGATTAAAACACTGAACTGACAATTAACAGCCCAATATCTACAATCAACCAAC
AAGTCATTATTACCCTCACTGTCAACCCAACACAGGCATGCTCATAAGGAAAGGTTAAAA
AAAGTAAAAGGAACTCGGCAAATCTTACCCCGCCTGTTTACCAAAAACATCACCTCTAGC
ATCACCAGTATTAGAGGCACCGCCTGCCCAGTGACACATGTTTAACGGCCGCGGTACCCT
AACCGTGCAAAGGTAGCATAATCACTTGTTCCTTAAATAGGGACCTGTATGAATGGCTCC
```

Sequence differences occur because of...

1. Sequencing error
2. Genetic variation

Approximate matching

```
T: GGAAAAAGAGGTAGCGGCGTTTAACAGTAG
             ||| |||||
P:           GTAACGGCG
                ↑
             Mismatch
          (Substitution)
```

Approximate matching

```
T: G G A A A A A G A G G T A G C - G C G T T T A A C A G T A G
                       | | | | |   | | |
                    P: G T A G C G G C G
                                 ↑
                             Insertion
```

Approximate matching

```
T: G G A A A A A G A G G T A G C G G C G T T T A A C A G T A G
                       | |   | | | | | |
                    P: G T - G C G G C G
                           ↑
                        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

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

(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

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
   | |   | | | | | | | | | |   | | | |        Edit distance = 2
Y: T G A C C G C G C A A A A - C A G C
```

```
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

(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
   | |   | | | | | | | | | |   | | | |      Edit distance = 2
Y: T G A C C G C G C A A A A - C A G C

X: G C G T A T G C G G C T A - A C G C
   | |   | | | | | | | | | |   | | |        Edit distance = 2
Y: G C - T A T G C G G C T A T A C G C
```

Approximate matching

Like exact matching, but *pattern P* may be within a certain *distance* (usually Hamming or edit) of *T*. Each such place is an *approximate match*.

Allowing edits is more challenging than just allowing mismatches

We'll return to edits

Approximate matching

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

Approximate matching

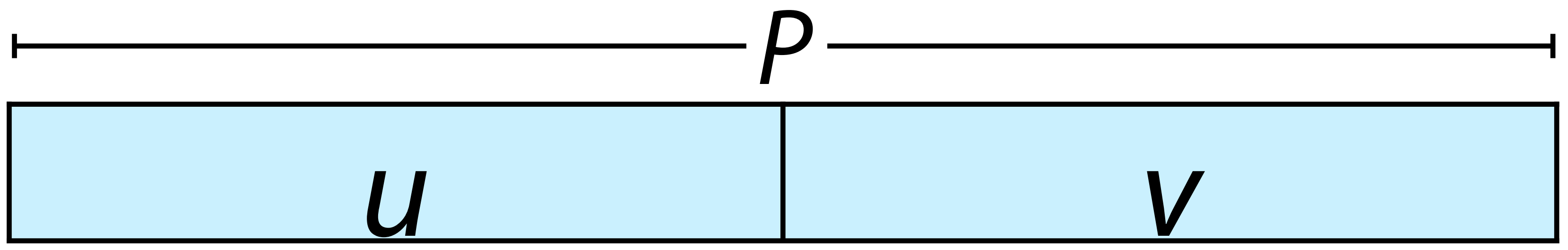
```
def naive_approx_hamming(p, t, maxDistance):
    occurrences = []
    for i in range(len(t) - len(p) + 1):  # loop over alignments
        nmm = 0
        for j in range(len(p)):           # loop over characters
            if t[i+j] != p[j]:            # compare characters
                nmm += 1                  # mismatch
                if nmm > maxDistance:
                    break                 # exceeded max hamming dist
        if nmm <= maxDistance:
          occurrences.append(i)           # approximate match
    return occurrences
```

http://bit.ly/CG_NaiveApprox

Approximate matching

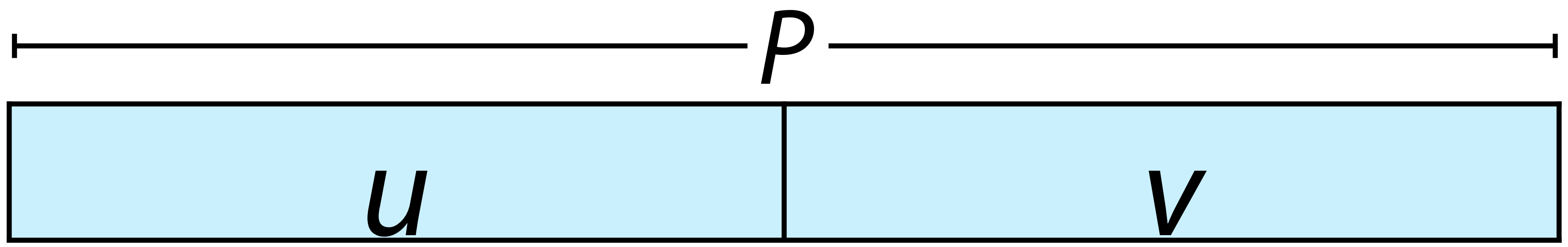
Wanted: way to apply exact matching algorithms to approximate matching problems

Approximate matching

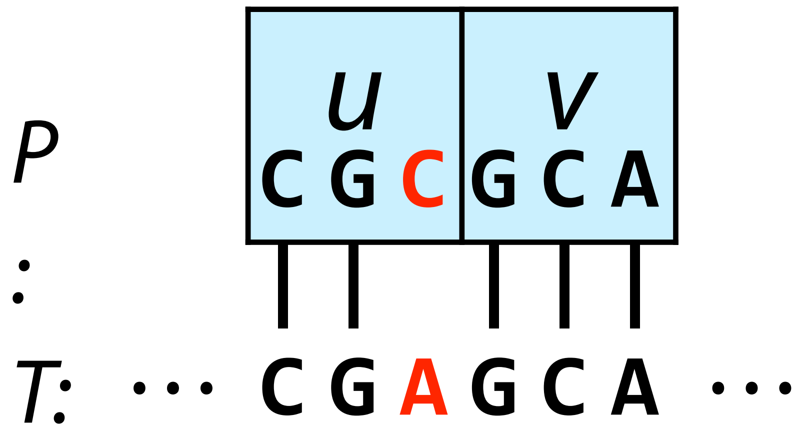


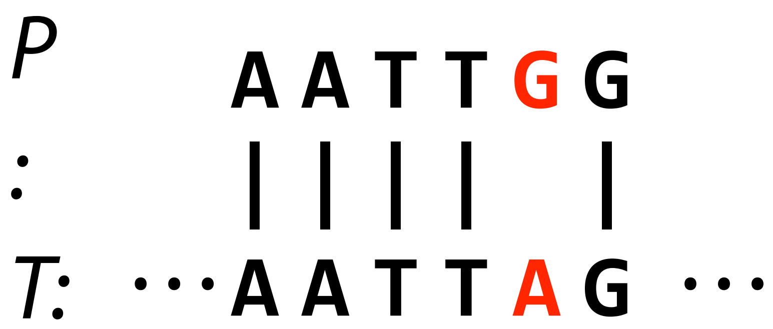
If *P* occurs in *T* with 1 edit, then *u* or *v* appears with no edits

Approximate matching

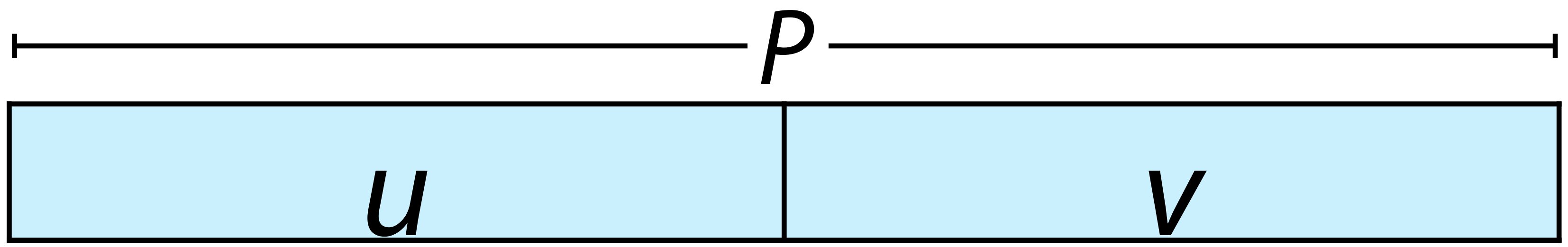


If *P* occurs in *T* with 1 edit, then *u* or *v* appears with no edits

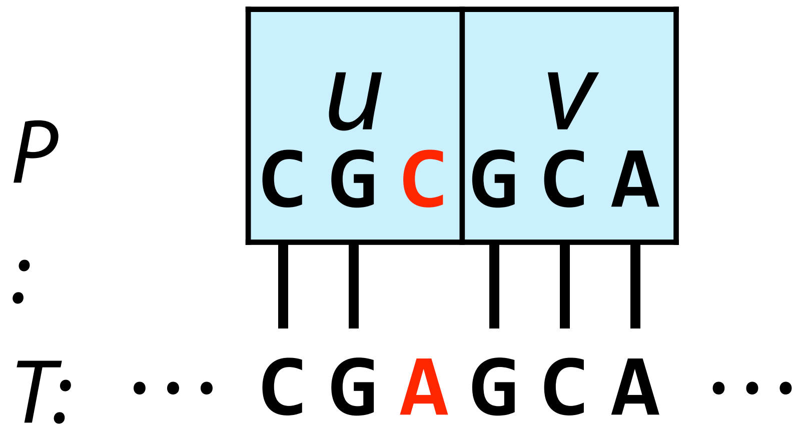


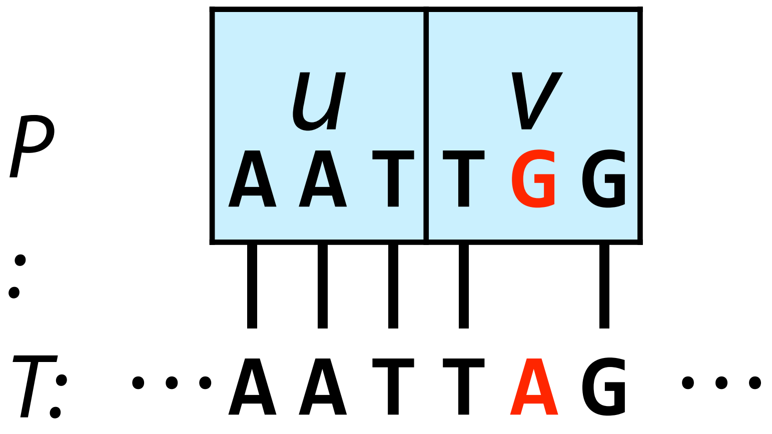


Approximate matching



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

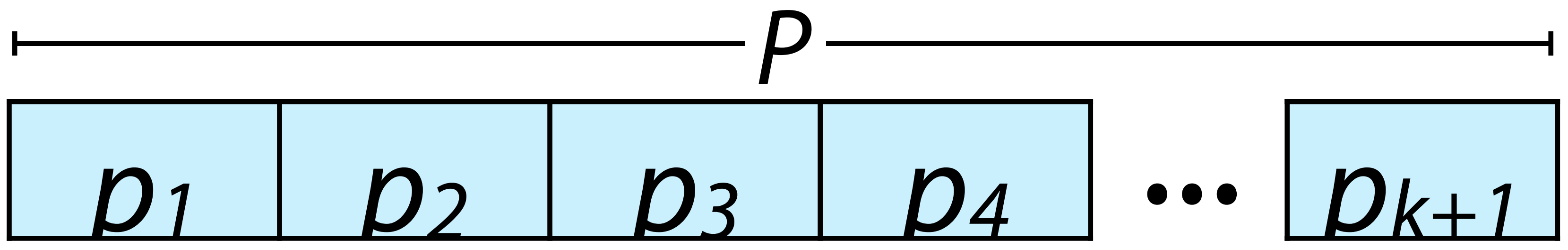




Approximate matching

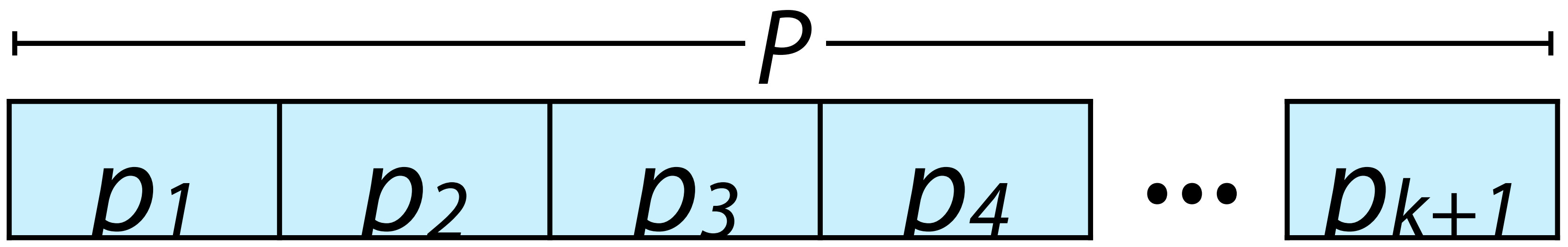
If *P* occurs in *T* with up to *k* edits…

Approximate matching



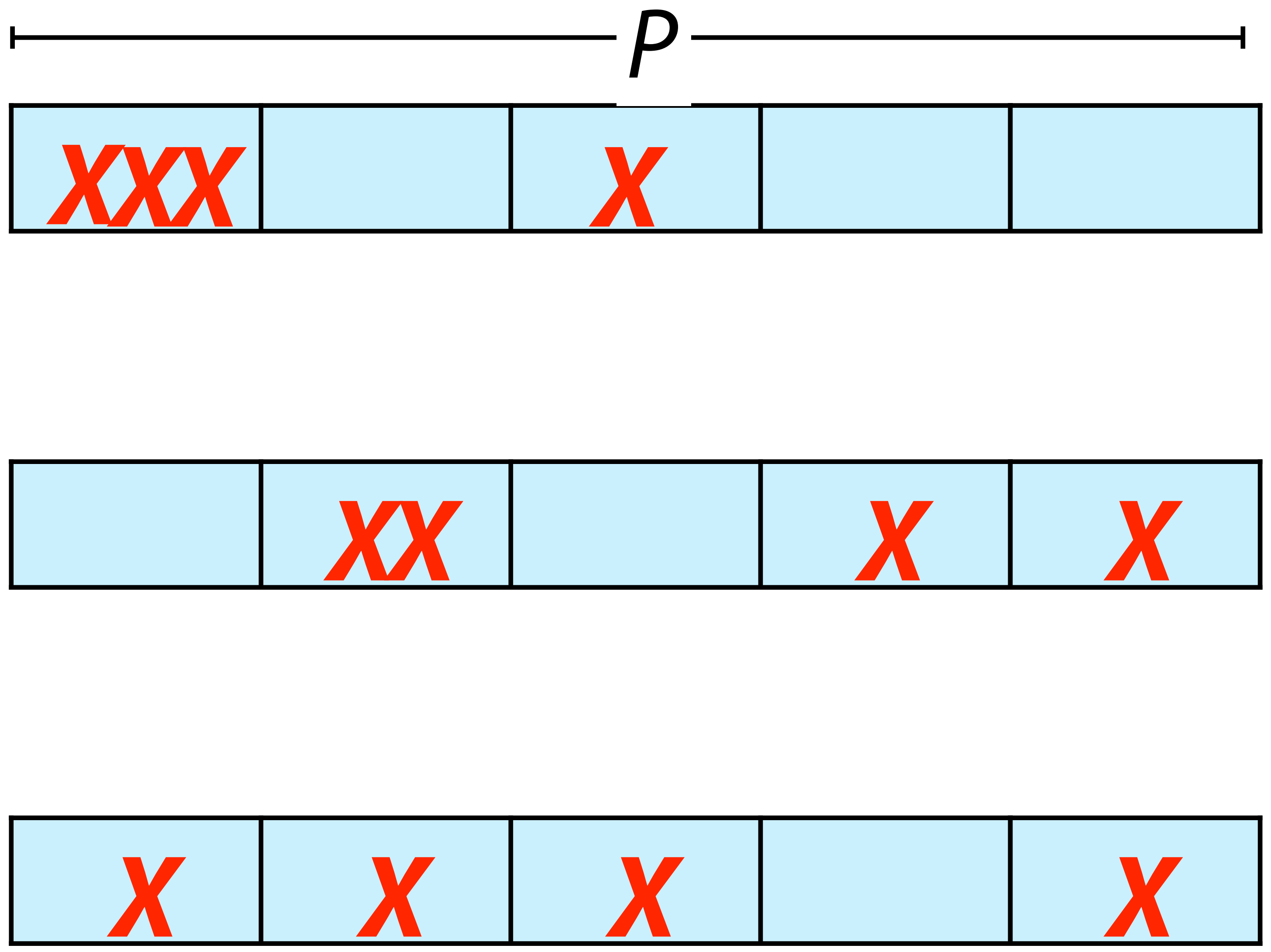
If *P* occurs in *T* with up to *k* edits…

Approximate matching



If P occurs in T with up to k edits, then at least one of p_1, p_2, ..., 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!

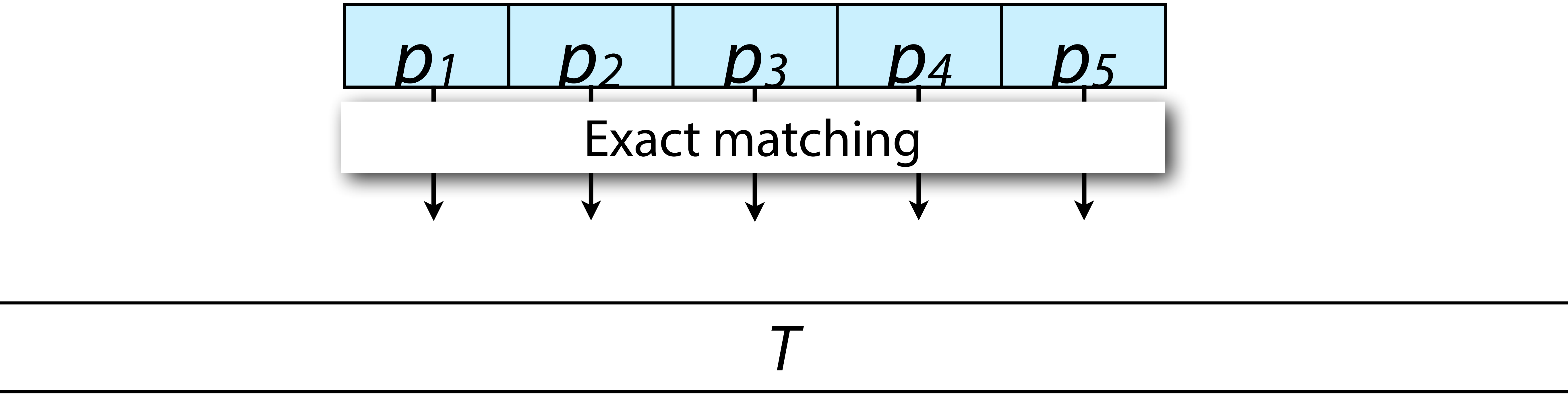
Approximate matching

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

Approximate matching

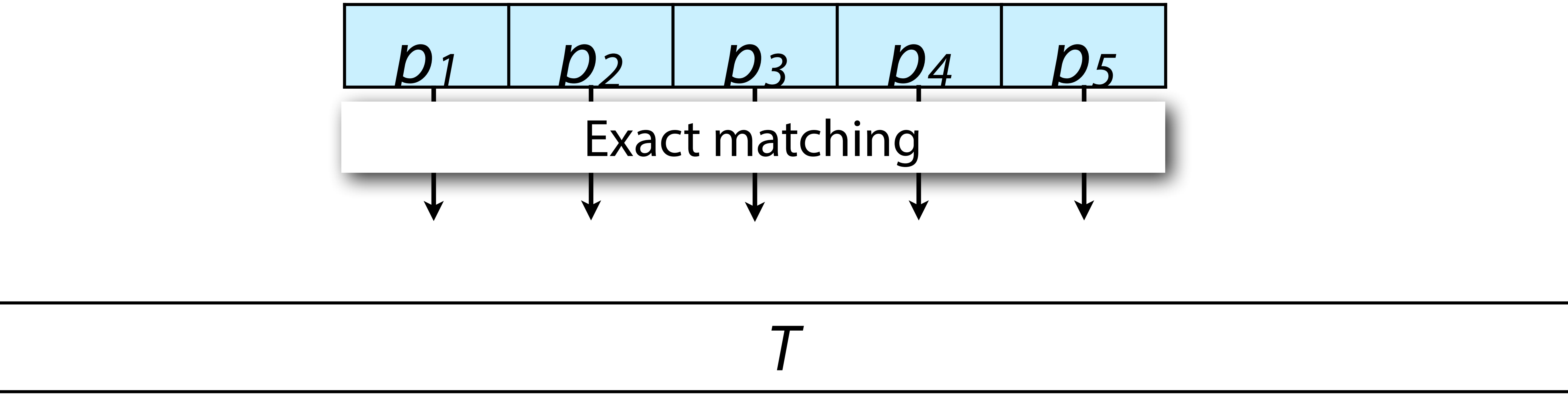
We have *k* pigeons, *k*+1 holes, at least one...
...is *empty*

Pigeonhole principle



What algorithm can we use?

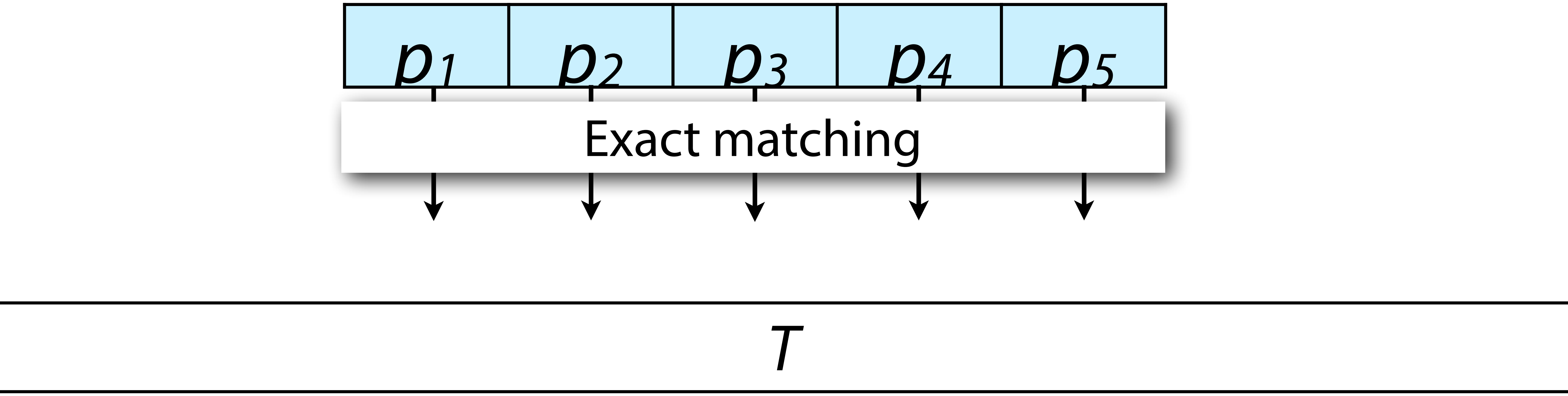
Pigeonhole principle



What algorithm can we use?

Any exact matching algorithm

Pigeonhole principle

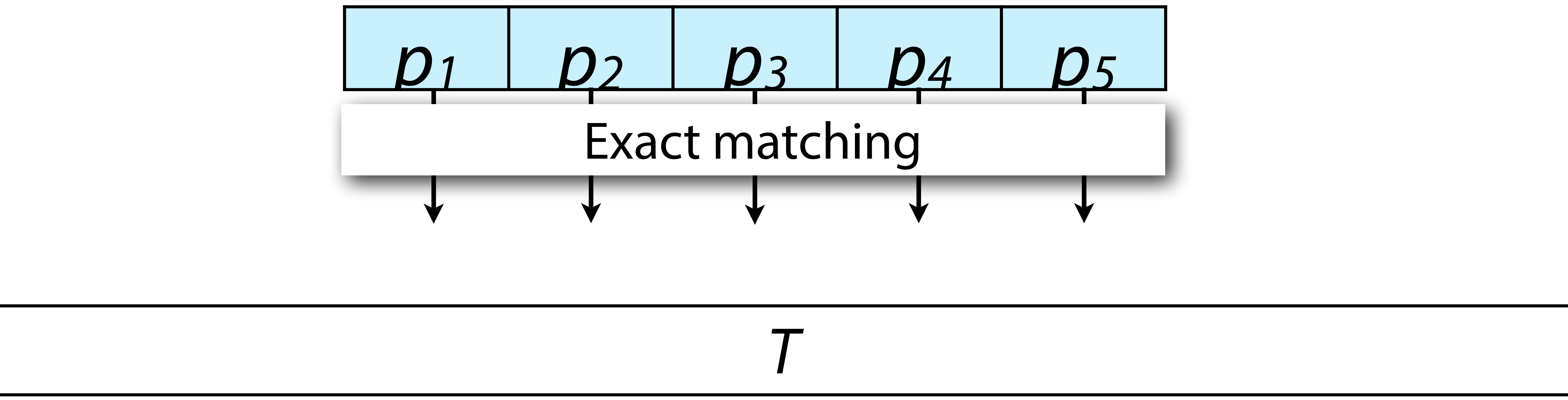


What algorithm can we use?

Any exact matching algorithm

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

Pigeonhole principle



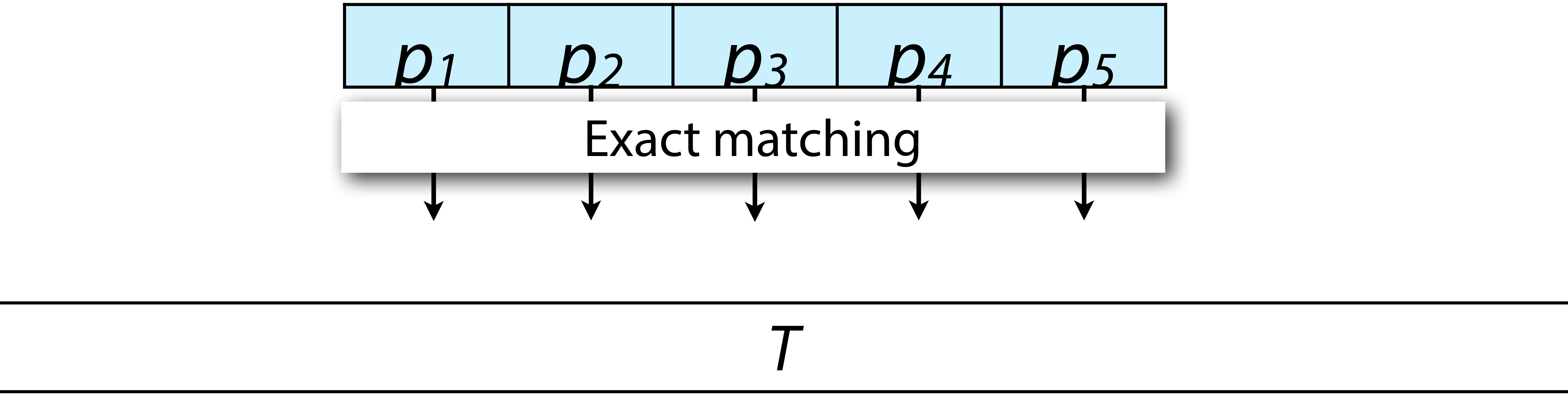
What algorithm can we use?

Any exact matching algorithm

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

Naive exact matching

Pigeonhole principle



What algorithm can we use?

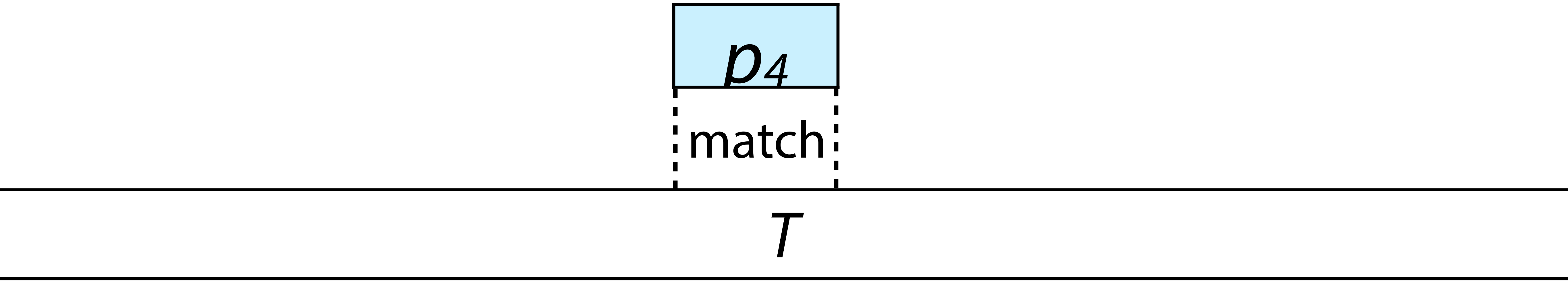
Any exact matching algorithm

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

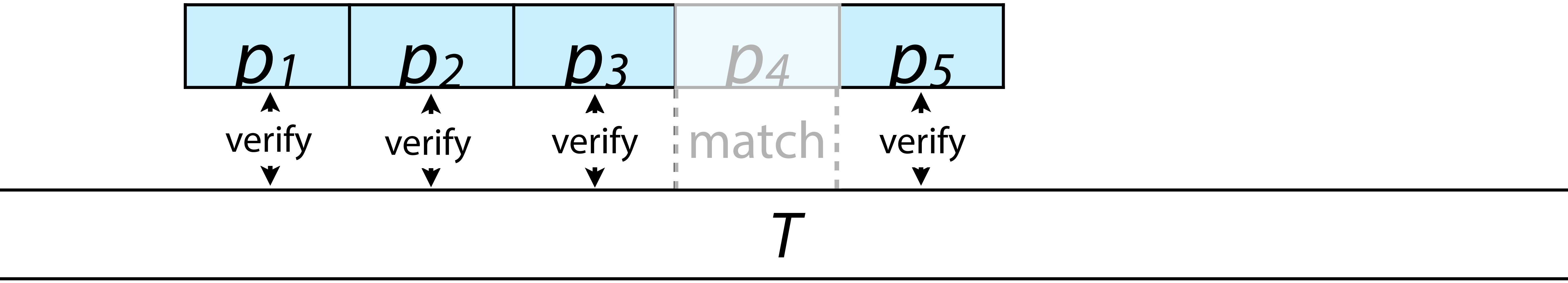
Naive exact matching

Boyer-Moore

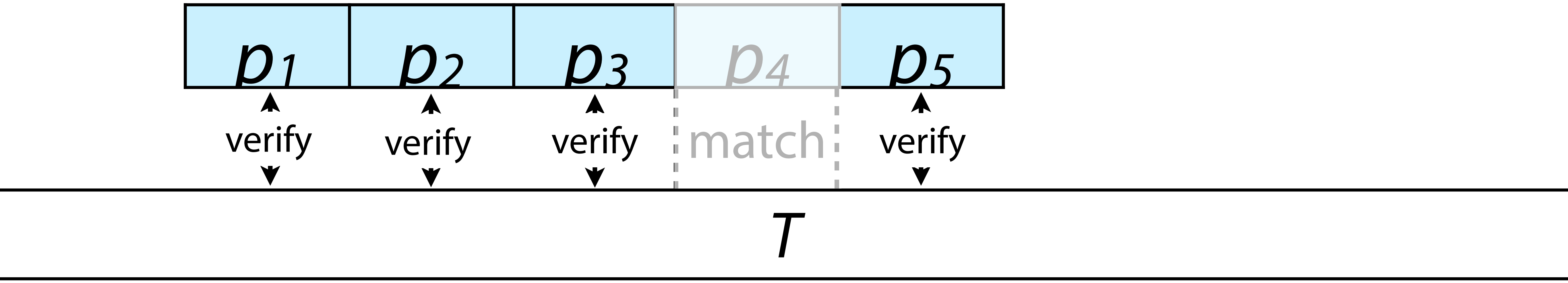
Pigeonhole principle



Pigeonhole principle

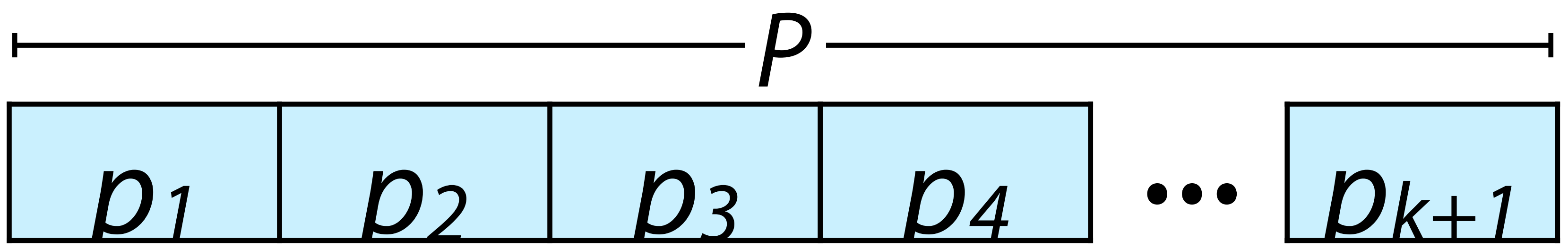


Pigeonhole principle



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

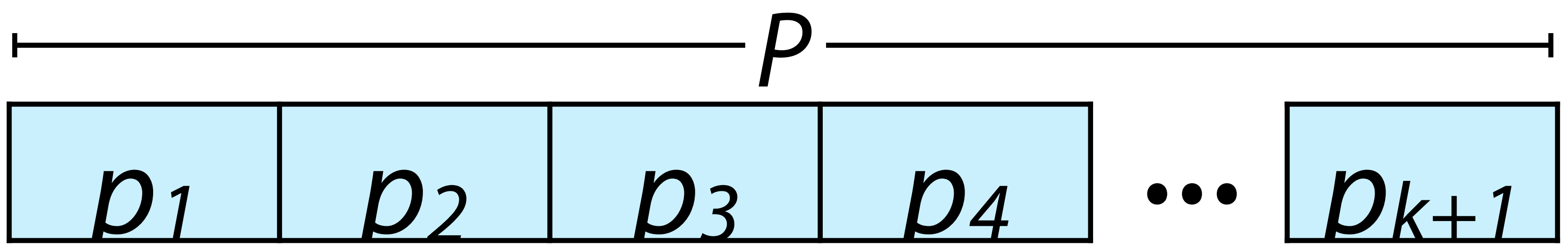
Pigeonhole principle



Advantages

Disadvantages

Pigeonhole principle

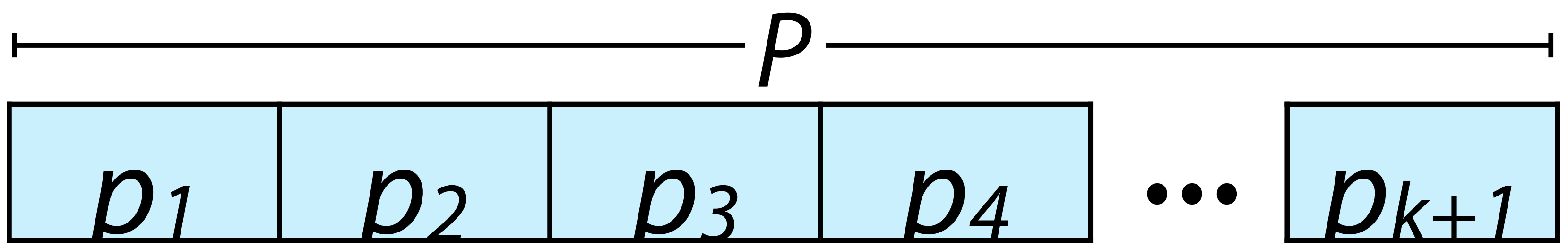


Advantages

Reuse favorite exact matching algos; fast and easy

Disadvantages

Pigeonhole principle



Advantages

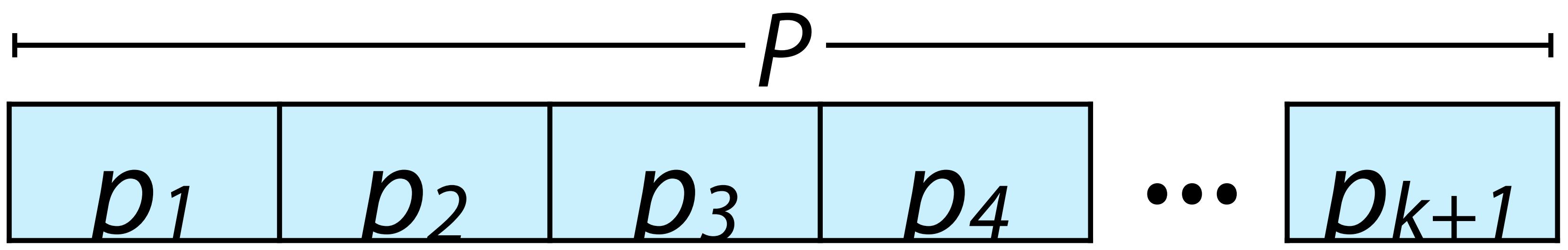
Reuse favorite exact matching algos; fast and easy

Flexible; works for Hamming and edit distance*

Disadvantages

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

Pigeonhole principle



Advantages

Reuse favorite exact matching algos; fast and easy

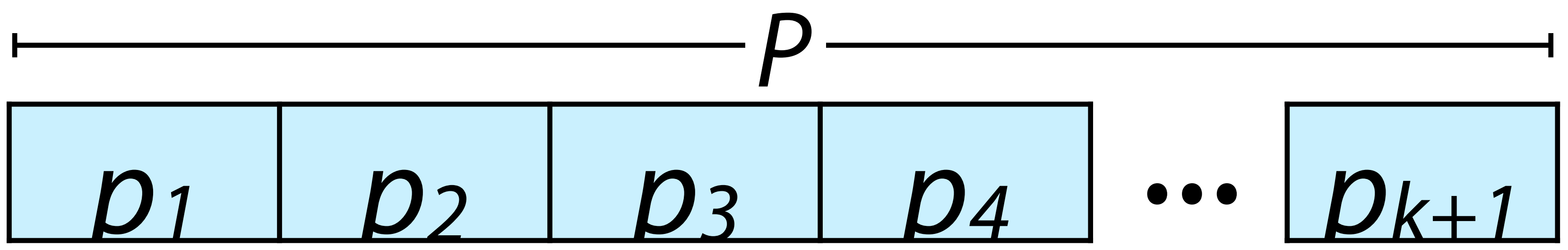
Flexible; works for Hamming and edit distance*

Disadvantages

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

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

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

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	1.91s	17						
P: 50 nt string from Alu repeat* **T**: Human reference (hg19) chromosome 1	32.5 M	67.21 s	336						

* GCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGG

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	1.91s	17	3.05 M	7.73 s	24			
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			

* GCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGG

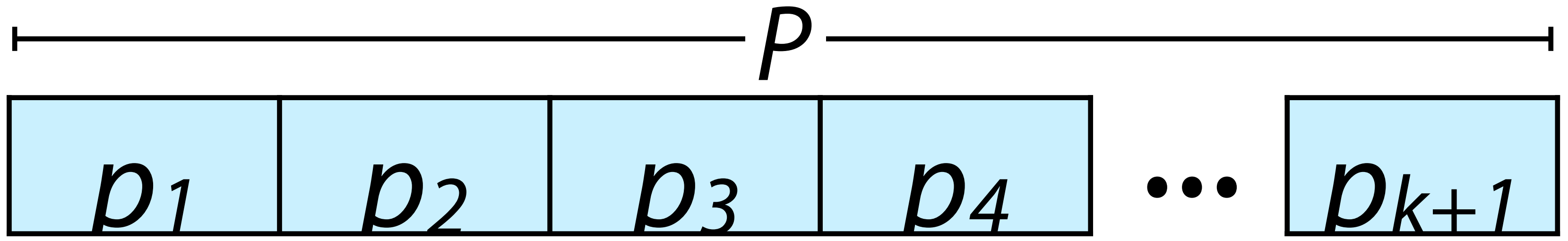
Implementation of pigeonhole principle with Boyer-Moore as exact matching algorithm: http://j.mp/CG_ApproxBM

	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	1.91s	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

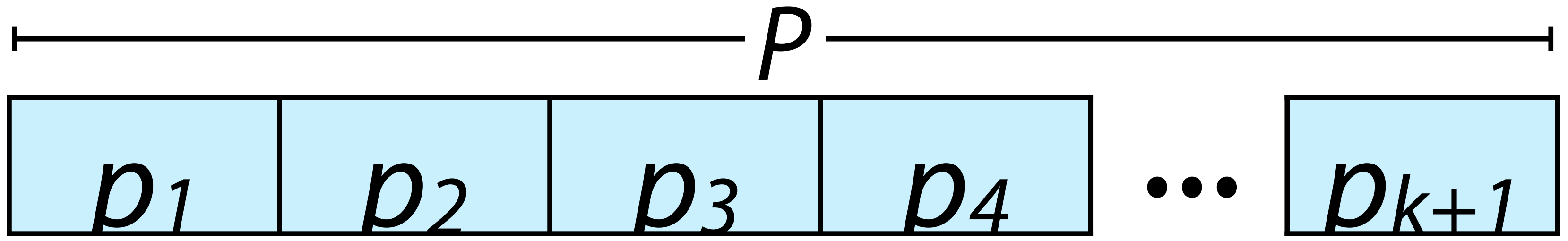
Generalizing pigeonhole, part 1

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



Generalizing pigeonhole, part 1

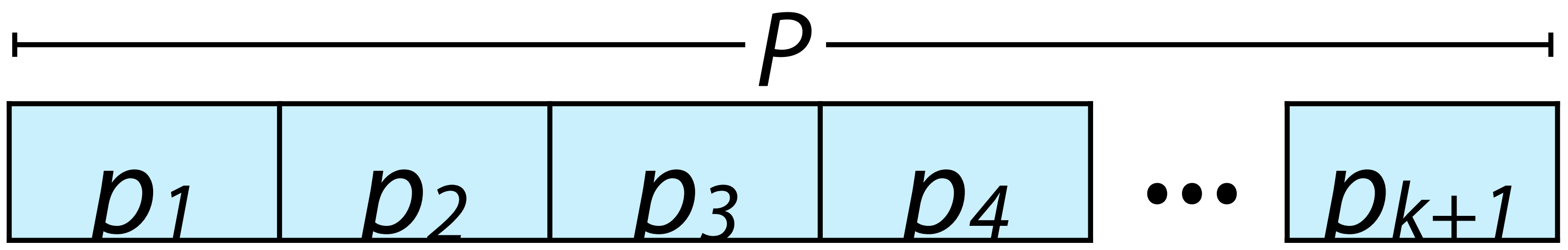
If *P* occurs in *T* with up to *k* edits, then at least one of p_1, p_2, ..., p_{k+1} must appear with 0 edits



But doesn't *have to* be "at least one of" …

Generalizing pigeonhole, part 1

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

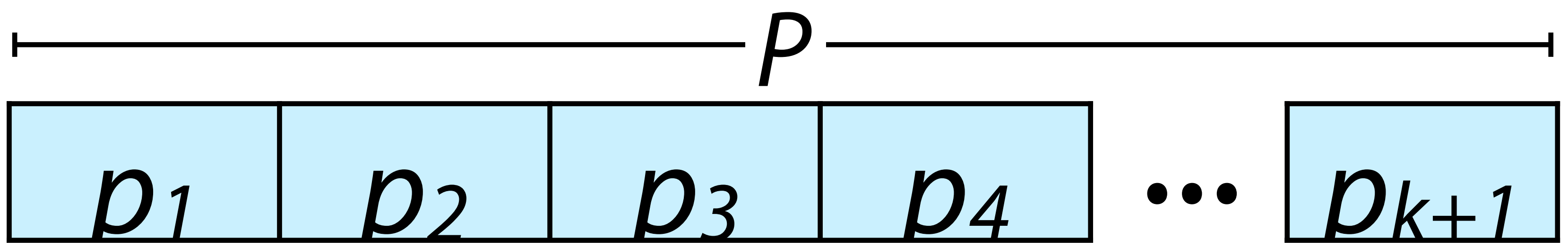


But doesn't *have to* be "at least one of" …

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

Generalizing pigeonhole, part 1

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



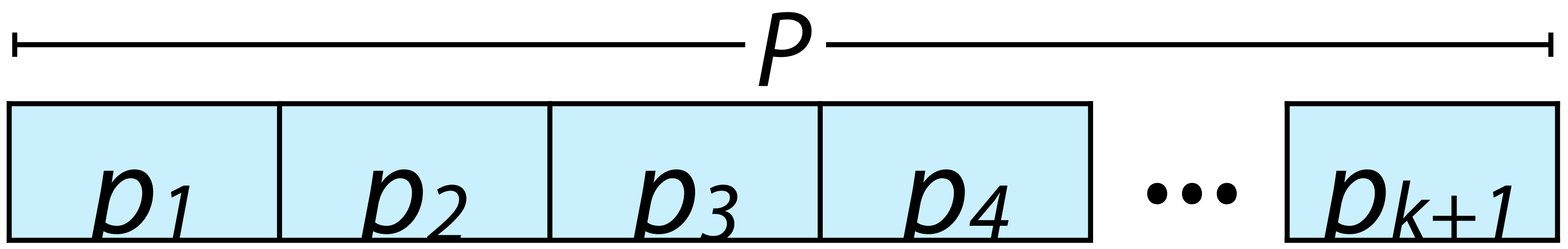
But doesn't *have to* be "at least one of" …

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, ..., p_{k+1} must appear with 0 edits



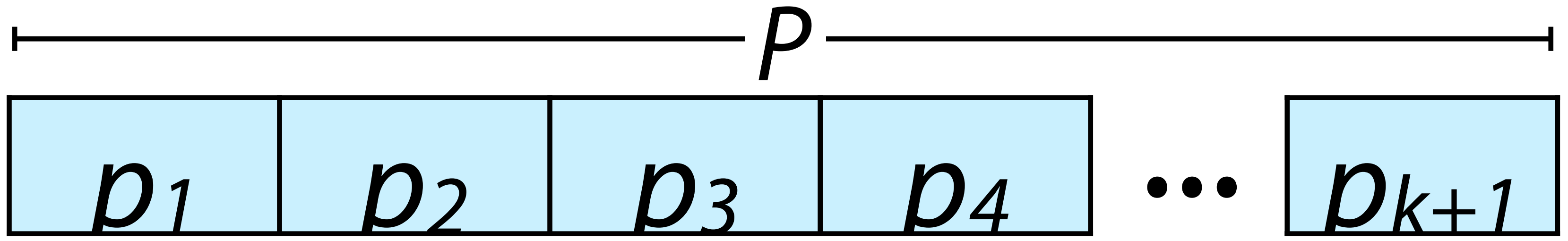
But doesn't *have to* be "at least one of" …

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, ..., p_{k+2} must appear with 0 edits

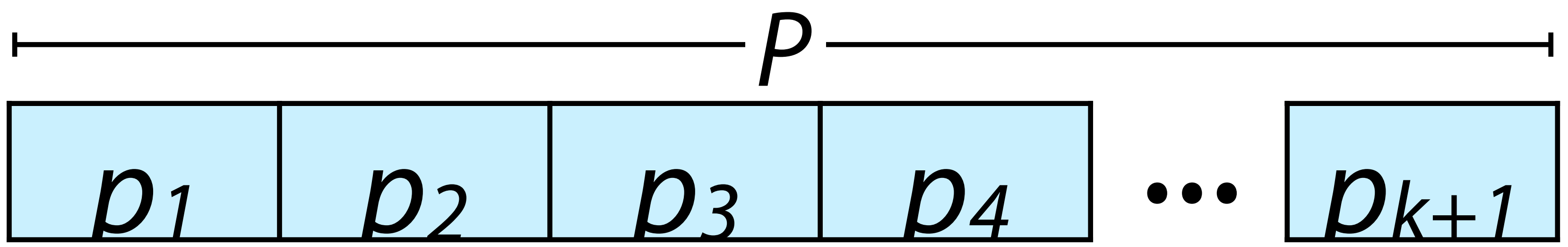
Generalizing pigeonhole, part 2

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

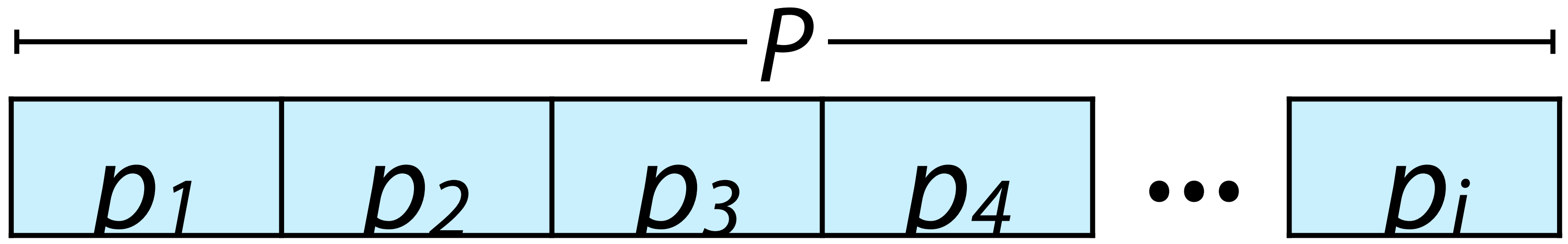


Generalizing pigeonhole, part 2

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

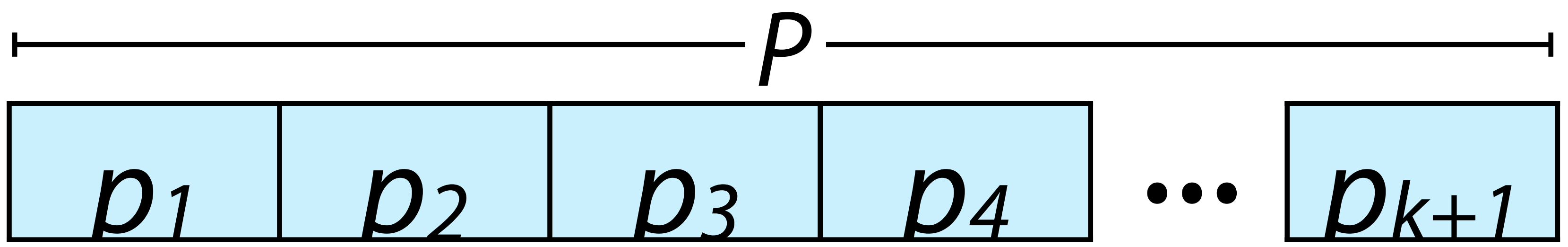


Let $p_1, p_2, ..., p_j$ be a partitioning of *P*. If *P* occurs with up to *k* edits, then at least one of $p_1, p_2, ..., p_j$ must occur with ≤ ??? edits.

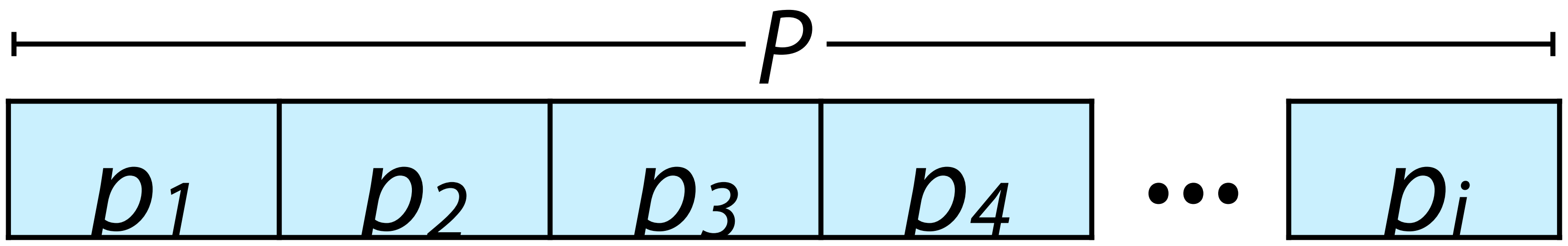


Generalizing pigeonhole, part 2

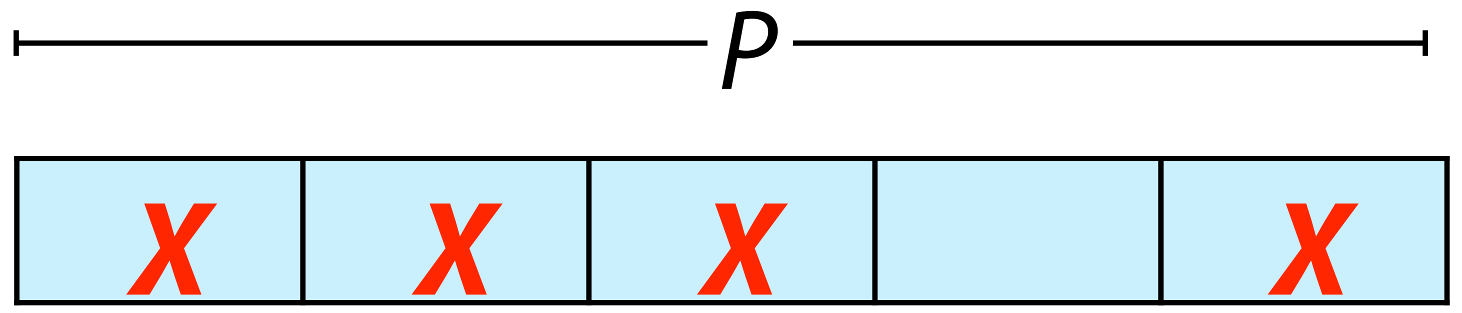
If *P* occurs in *T* with up to *k* edits, then at least one of $p_1, p_2, ..., p_{k+1}$ must appear with 0 edits



Let $p_1, p_2, ..., p_j$ be a partitioning of *P*. 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.



Generalizing pigeonhole, part 2

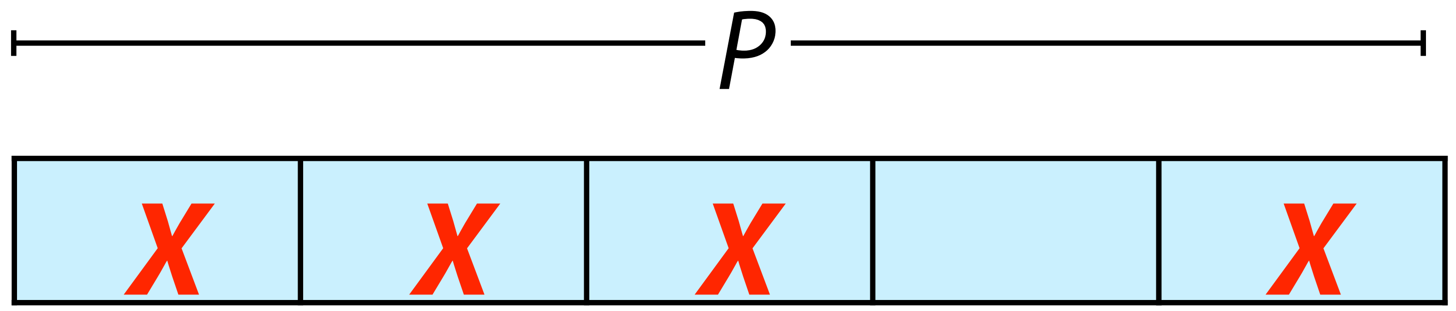


k = 4 edits

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

Generalizing pigeonhole, part 2

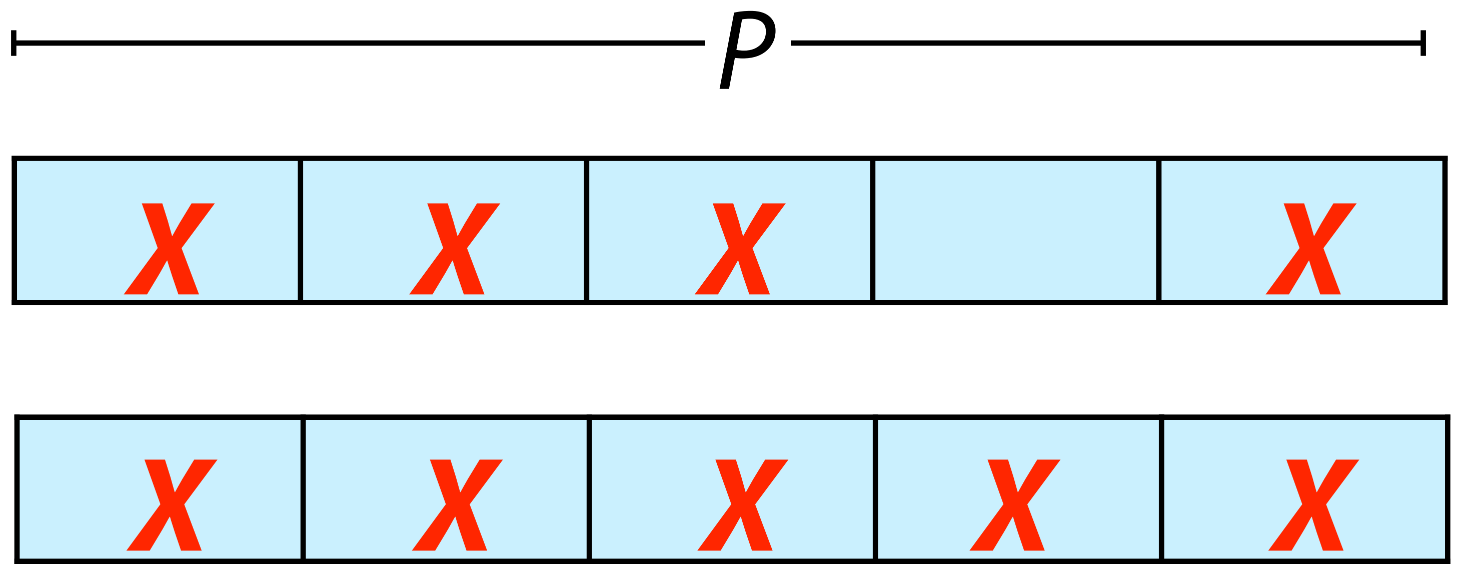
At least one of $p_1, p_2, ..., p_5$ occurs with...



k = 4 edits $\leq$0 edits

Generalizing pigeonhole, part 2

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

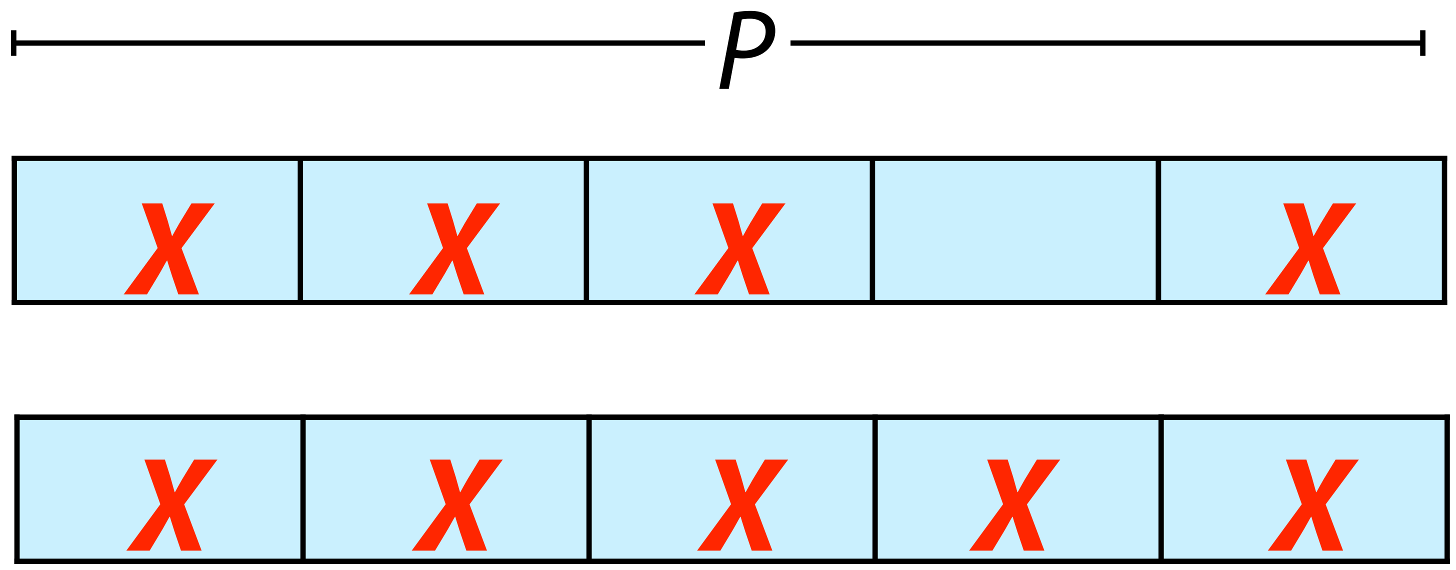


$k = 4$ edits ≤0 edits

$k = 5$ edits

Generalizing pigeonhole, part 2

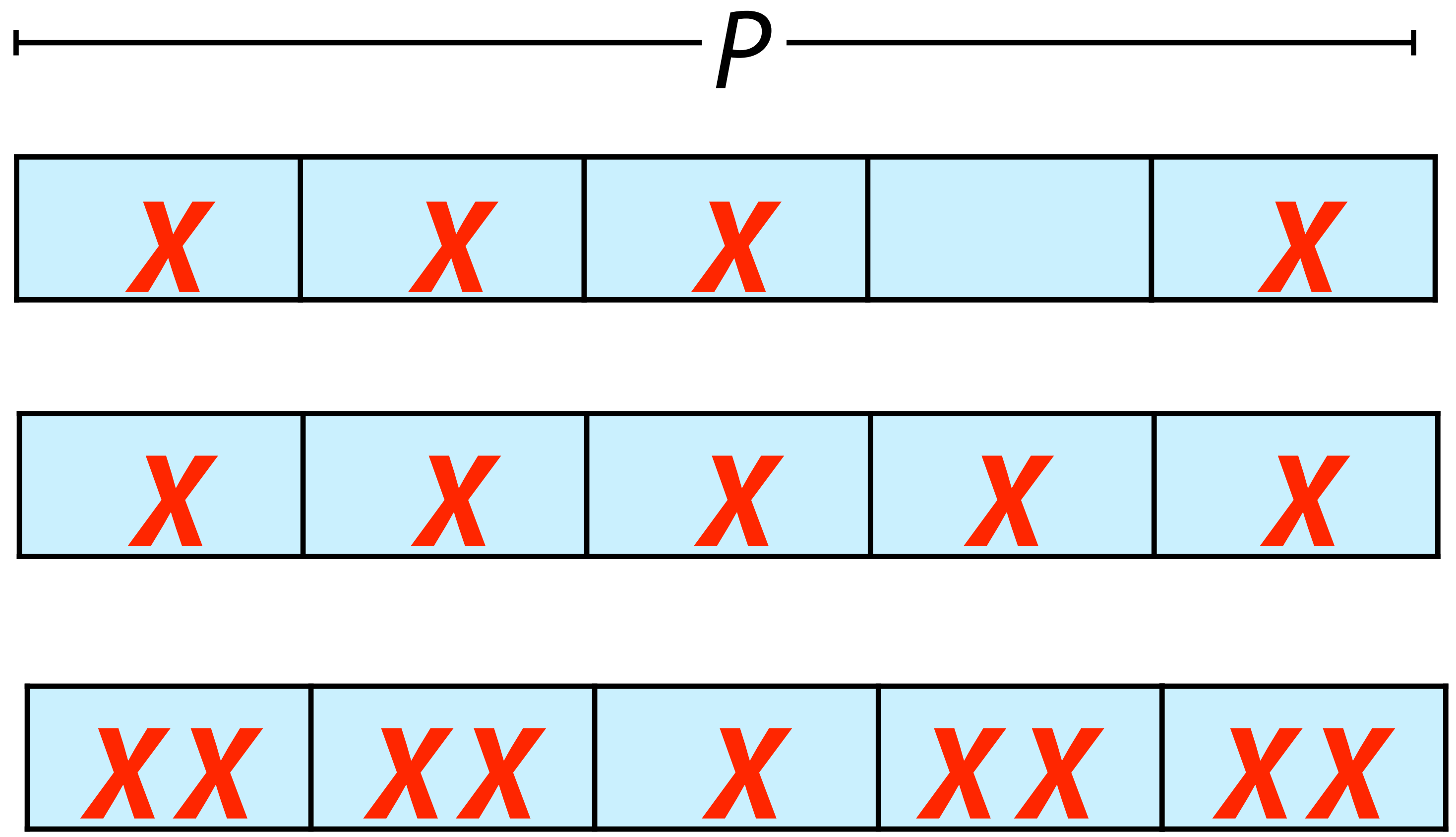
At least one of $p_1, p_2, \ldots, p_5$ occurs with...



k = 4 edits	≤0 edits
k = 5 edits	≤1 edits

Generalizing pigeonhole, part 2

At least one of $p_1, p_2, ..., p_5$ occurs with...



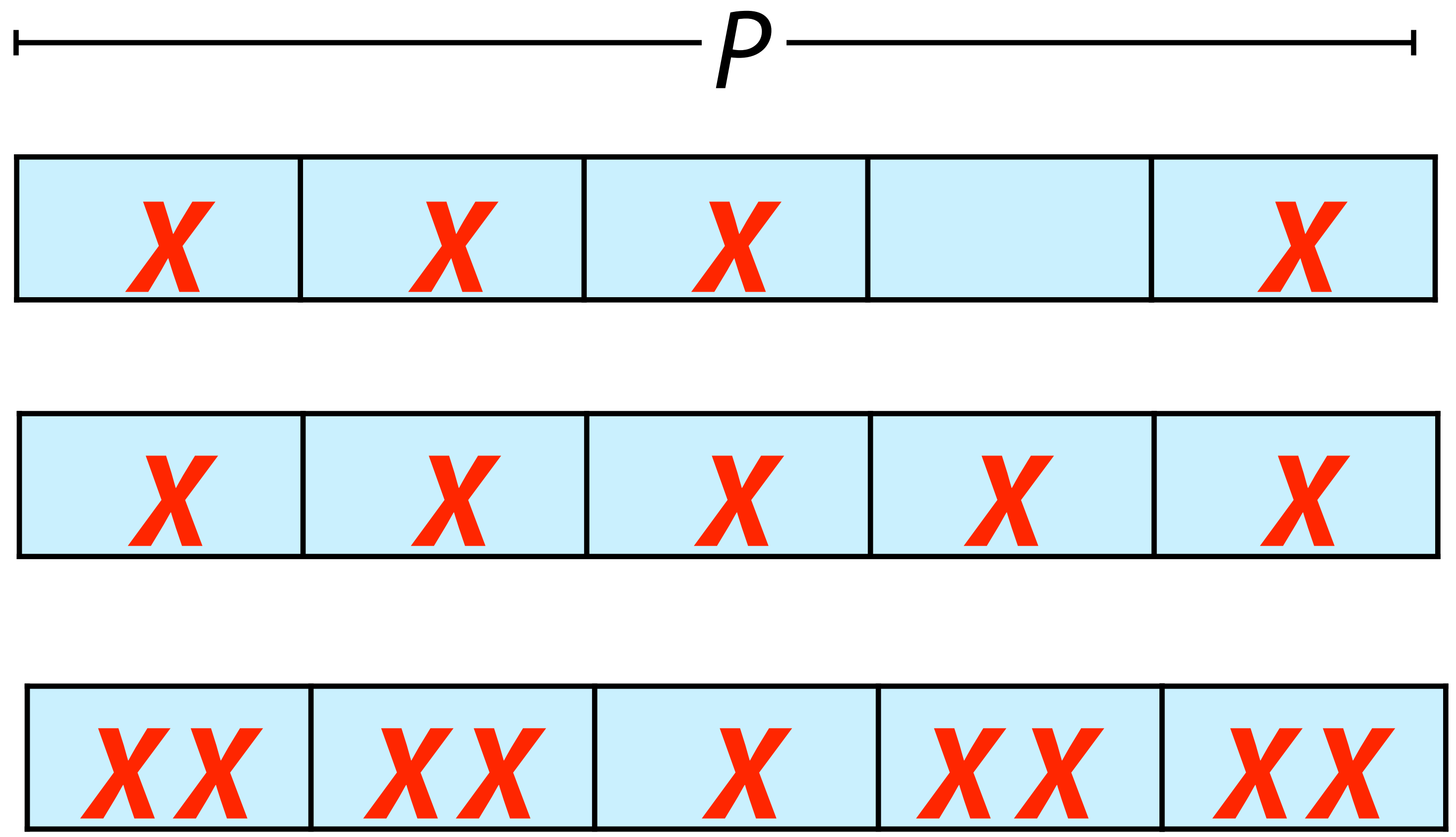
k = 4 edits ≤0 edits

k = 5 edits ≤1 edits

k = 9 edits

Generalizing pigeonhole, part 2

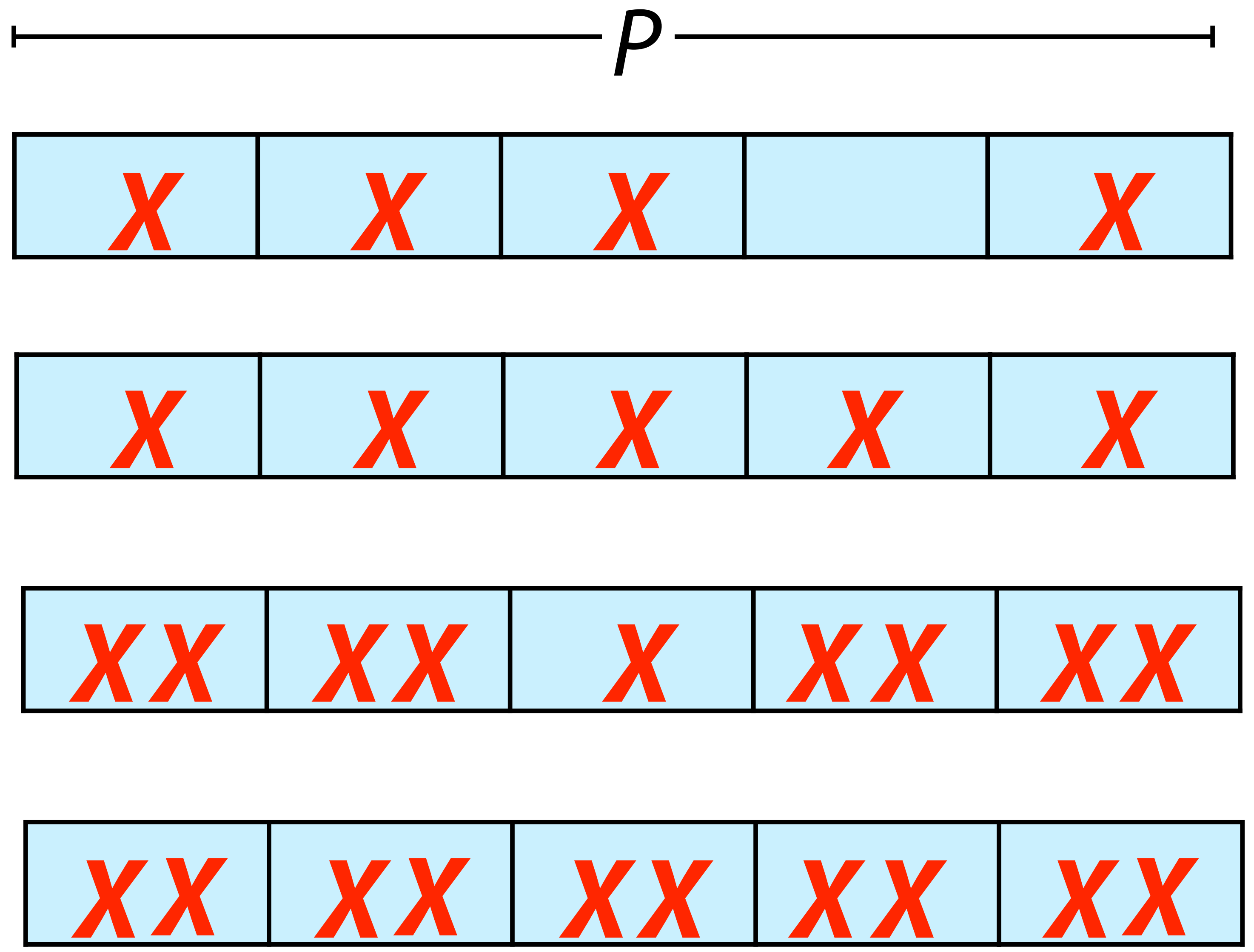
At least one of $p_1, p_2, \ldots, p_5$ occurs with...



k = 4 edits	≤0 edits
k = 5 edits	≤1 edits
k = 9 edits	≤1 edits

Generalizing pigeonhole, part 2

At least one of $p_1, p_2, ..., p_5$ occurs with...



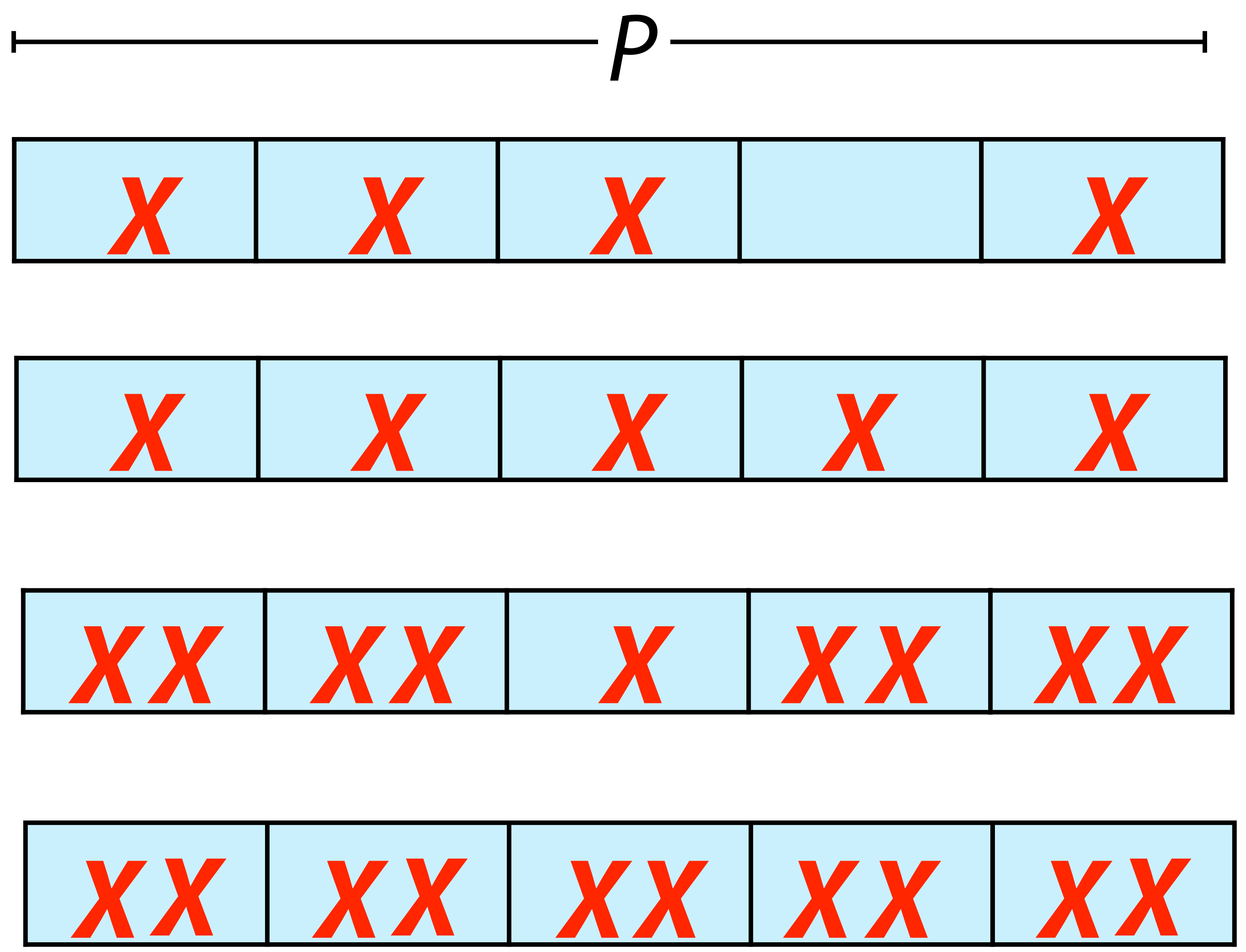
k = 4 edits ≤0 edits

k = 5 edits ≤1 edits

k = 9 edits ≤1 edits

k = 10 edits

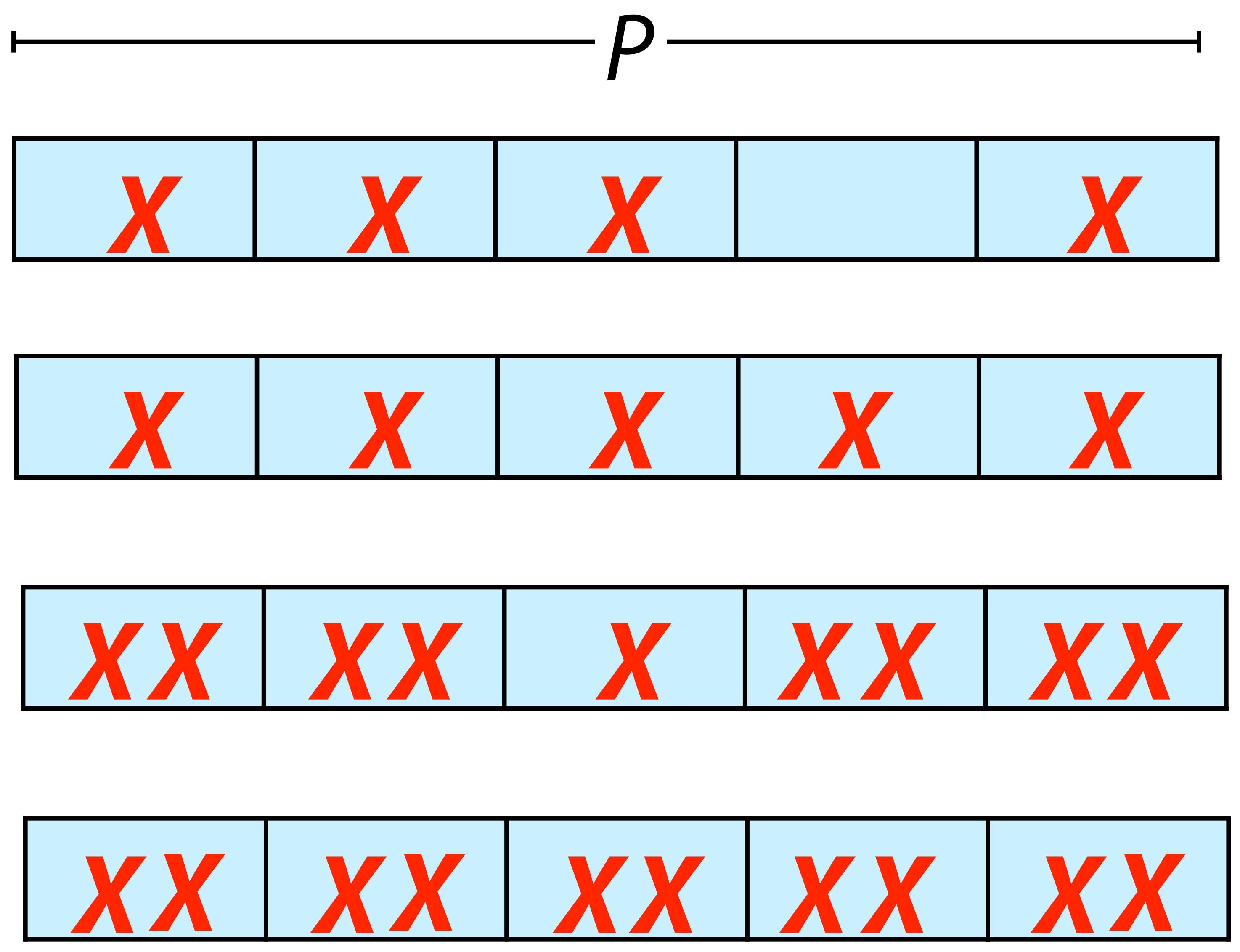
Generalizing pigeonhole, part 2



	At least one of $p_1, p_2, ..., p_5$ 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

At least one of $p_1, p_2, ..., p_5$ occurs with...



k = 4 edits	≤0 edits
k = 5 edits	≤1 edits
k = 9 edits	≤1 edits
k = 10 edits	≤2 edits

etc

Generalizing pigeonhole, part 2

General	*Pigeonhole principle* If *P* occurs in *T* with up to *k* edits, then at least one of $p_1, p_2, ..., p_j$ must appear with *floor*(k / j) edits
Specific	*Pigeonhole principle with $j = k + 1$* If *P* occurs in *T* with up to *k* edits, then at least one of $p_1, p_2, ..., p_{k+1}$ must appear with 0 edits

Generalizing pigeonhole, part 2

General

Pigeonhole principle

If P occurs in T with up to k edits, then at least one of $p_1, p_2, ..., p_j$ must appear with *floor*(k / j) edits

Specific

Pigeonhole principle with $j = k + 1$

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

Generalizing pigeonhole, part 2

Let $j = k + 1$

Why?

General

Pigeonhole principle

If P occurs in T with up to k edits, then at least one of $p_1, p_2, ..., p_j$ must appear with *floor*(k / j) edits

Specific

Pigeonhole principle with $j = k + 1$

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

Generalizing pigeonhole, part 2

Let $j = k + 1$

Why?
Smallest value s.t. *floor*(k / j) = 0

General

Pigeonhole principle

If P occurs in T with up to k edits, then at least one of p_1, p_2, ..., p_j must appear with *floor*(k / j) edits

Specific

Pigeonhole principle with $j = k + 1$

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

Generalizing pigeonhole, part 2

General

Pigeonhole principle

If *P* occurs in *T* with up to *k* edits, then at least one of $p_1, p_2, \ldots, p_j$ must appear with *floor*(*k* / *j*) edits

Specific

Pigeonhole principle with j = k + 1

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

Let *j* = *k* + 1

Why?

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

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

Generalizing pigeonhole, part 2

General

Pigeonhole principle

If P occurs in T with up to k edits, then at least one of $p_1, p_2, ..., p_j$ must appear with *floor*(k / j) edits

Specific

Pigeonhole principle with $j = k + 1$

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

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

Generalizing pigeonhole, part 2

General

Pigeonhole principle

If P occurs in T with up to k edits, then at least one of $p_1, p_2, ..., p_j$ must appear with *floor*(k / j) edits

Specific

Pigeonhole principle with $j = k + 1$

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

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?

Generalizing pigeonhole, part 2

General

Pigeonhole principle

If P occurs in T with up to k edits, then at least one of $p_1, p_2, ..., p_j$ must appear with *floor*(k / j) edits

Specific

Pigeonhole principle with $j = k + 1$

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

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

Generalizing pigeonhole, part 2

General

Pigeonhole principle

If P occurs in T with up to k edits, then at least one of $p_1, p_2, ..., p_j$ must appear with *floor*(k / j) edits

Specific

Pigeonhole principle with $j = k + 1$

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

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?

Generalizing pigeonhole, part 2

General

Pigeonhole principle

If P occurs in T with up to k edits, then at least one of $p_1, p_2, ..., p_j$ must appear with *floor*(k / j) edits

Specific

Pigeonhole principle with $j = k + 1$

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

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?

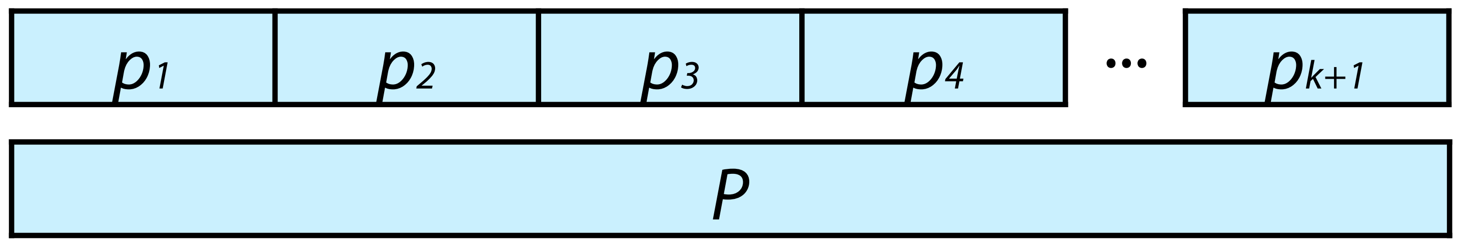
Yields fewer, longer partitions

Why are long partitions good?

Makes exact-matching filter more specific, minimizing # candidates

A different principle

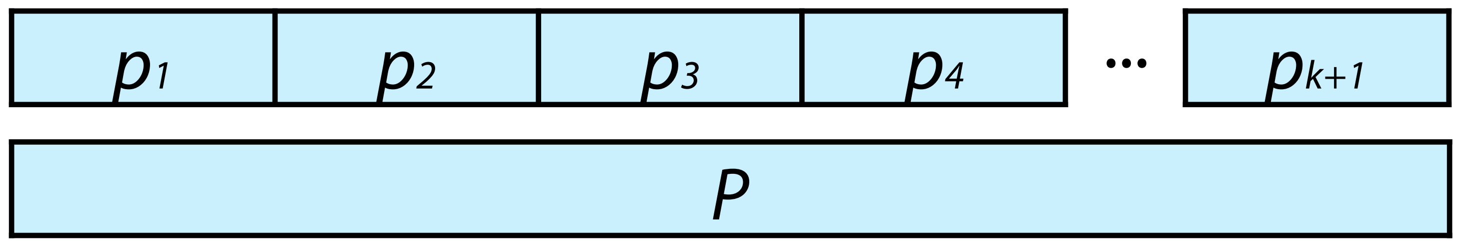
We partitioned *P* into non-overlapping substrings



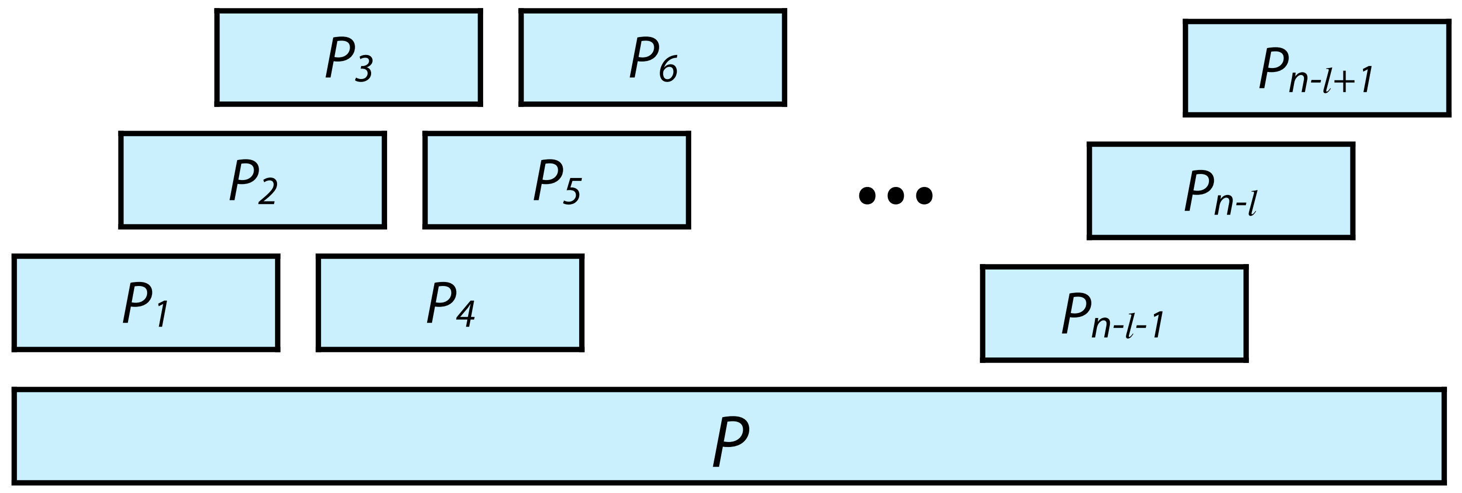
Now consider *overlapping* substrings

A different principle

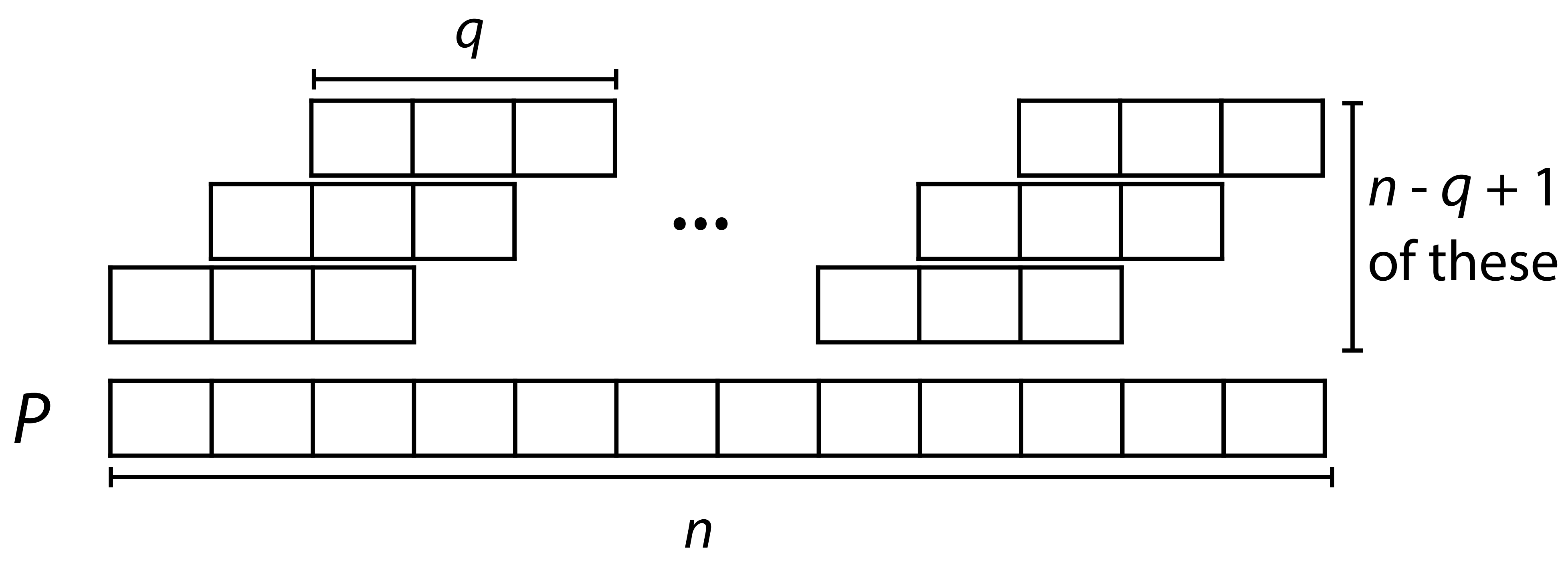
We partitioned *P* into non-overlapping substrings



Now consider *overlapping* substrings

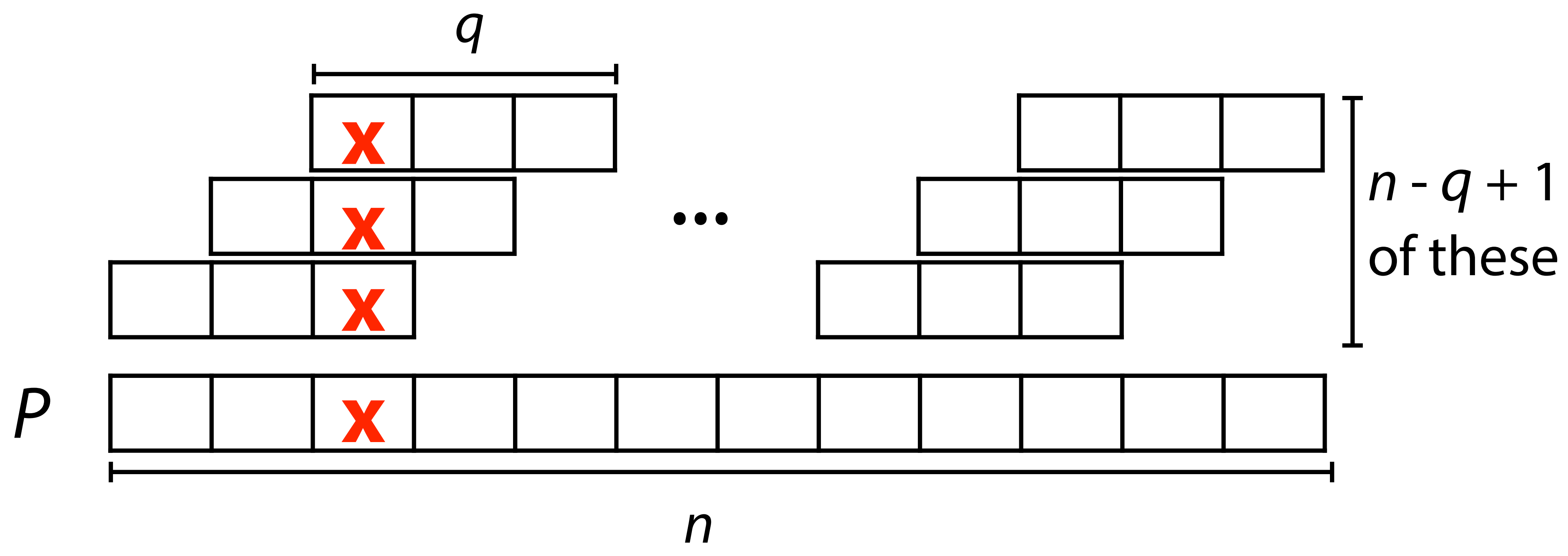


Approximate string matching: more



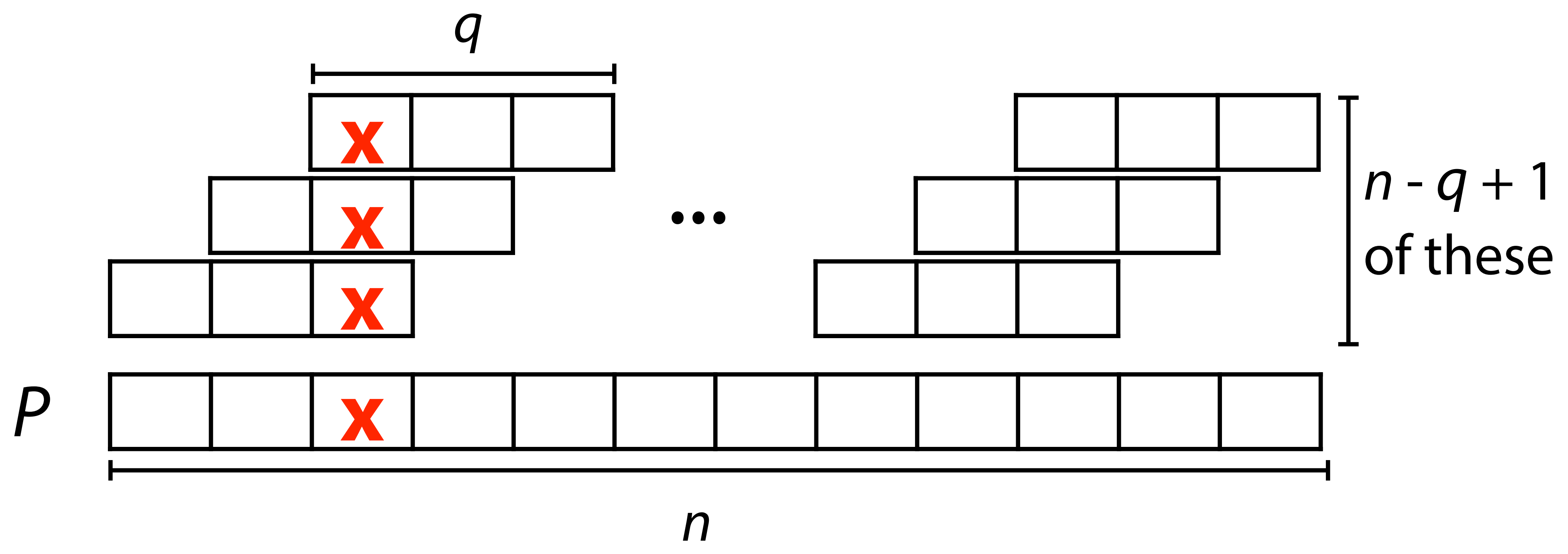
Say substrings are length *q*. There are *n* - *q* + 1 such substrings.

Approximate string matching: more



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

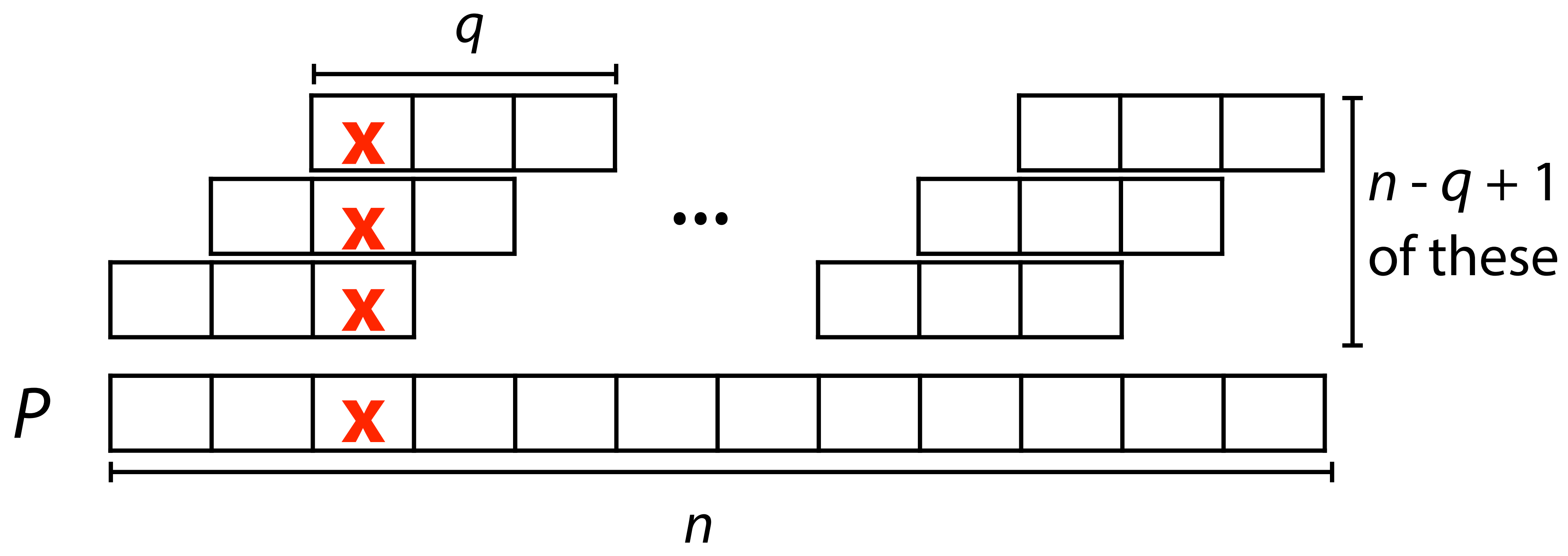
Approximate string matching: more



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

1 edit to P changes *at most q* substrings

Approximate string matching: more

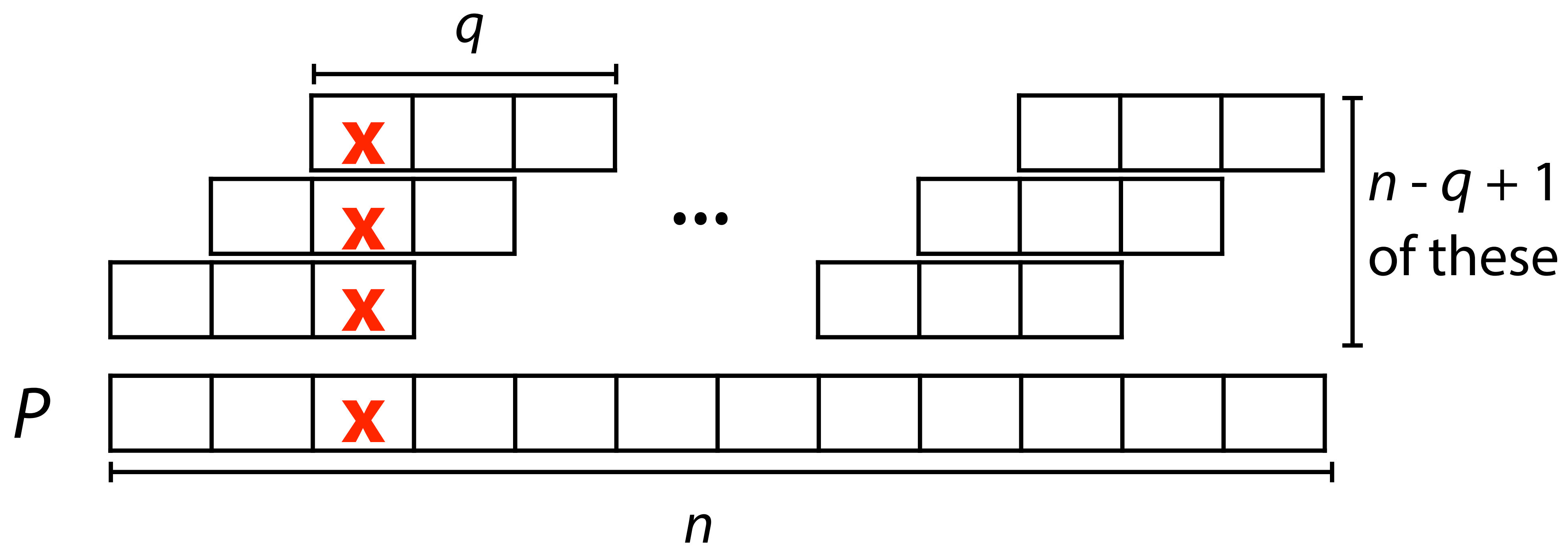


Say substrings are length q. There are $n - q + 1$ such substrings.

1 edit to P changes *at most* q substrings

Minimum # of length-q substrings unedited after k edits?

Approximate string matching: more



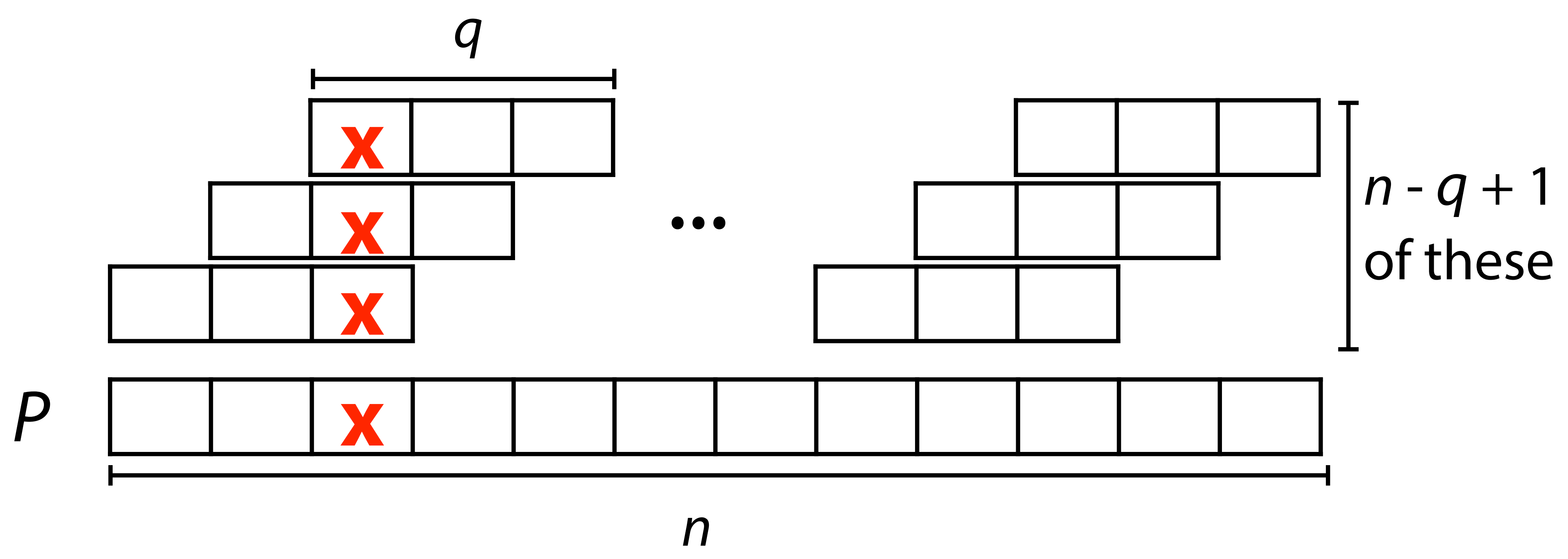
Say substrings are length q. There are $n - q + 1$ such substrings.

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? $n - q + 1 - \boxed{kq}$

Approximate string matching: more



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? $n - q + 1 - \boxed{kq}$

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

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

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

Approximate string matching: more

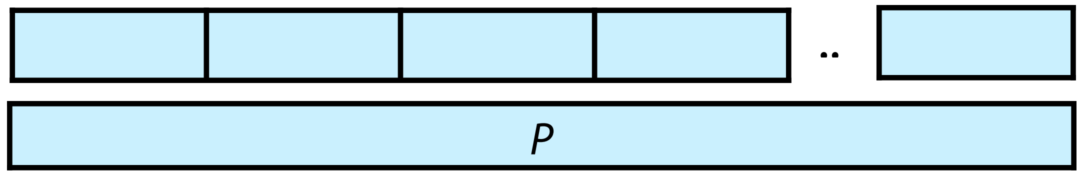
If P occurs in T with up to k edits, alignment contains an exact match of length q, where $q \geq \mathit{floor}(n / (k + 1))$

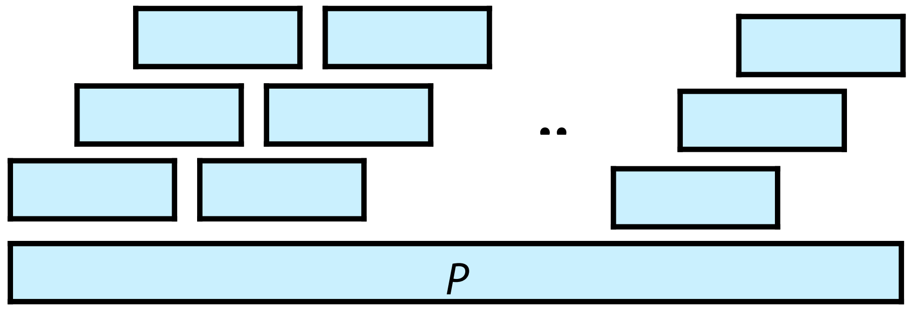
Obtained by solving for q: $n - q + 1 - kq \geq 1$

Exact matching filter: find matches of length $\mathit{floor}(n / (k + 1))$ 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 $\leq k$ 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 $j = k + 1$* $p_1, p_2, \ldots, p_{k+1}$ 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))



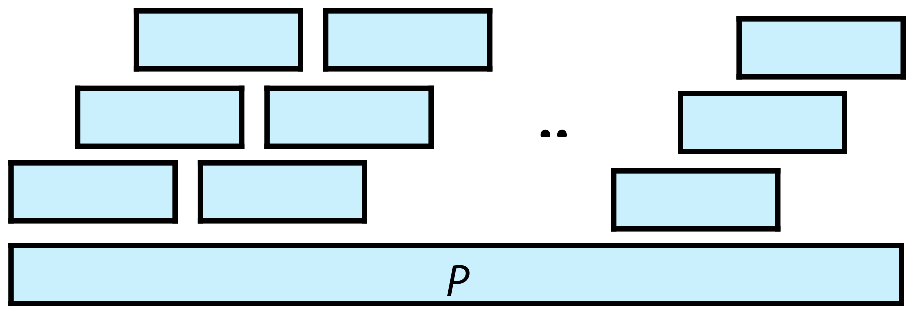


Sensitivity

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 usually much speedier & still acceptably sensitive.



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