String Algorithms and Data Structures
Approximate Pattern Matching

CS 199-225

Brad Solomon

November 11, 2024

UNIVERSITY OF
ILLINOIS
URBANA-CHAMPAIGN

Department of Computer Science

Learning Objectives

Review exact pattern matching and introduce approximate matching

Formally define a mismatch vs an edit

Discuss strategies for efficient approximate pattern matching…

… With mismatches

… With edits

Suffix-Based Index Review

	Suffix tree	**Suffix array**	**FM Index**
Time: Does P occur?	$O(n)$	$O(n \log m)$	$O(n)$
Time: Count k occurrences of P	$O(n + k)$	$O(n \log m)$	$O(n)$
Time: Report k locations of P	$O(n + k)$	$O(n \log m + k)$	$O(n + k)$
Space	$O(m)$	$O(m)$	$O(m)$
Needs T?	*yes*	*yes*	*no*
Bytes per input character	>15	~4	~0.5

$m = |T|$, $n = |P|$, k = # occurrences of P in T

Limitations of exact pattern matching


```
}                                          >  strong                    Aa  ab  .*   2 of 2      ↑  ↓  ≡  ✕

/**
 * Returns the number of alignments skipped by Boyer-Moore
 * In this instance, Boyer-Moore is *only* the strong bad character rule [and right-to-left scanning]
 *
 * Also modifies the outlist vector to contain the index positions of all exact matches of P in T.
 * If no match is found, modifies the vector to contain one value '[-1]'
 *
 * @param P A std::string object which holds the Pattern string.
 * @param T A std::string object which holds the Text string.
 * @param alpha A std::string object which holds the Alphabet string.
 * @param outList An std::vector<int> array (by reference) that can be modified to contain all matche
 *
 * @return An int counting the number of skipped alignments using bad character.
 */
int bmoore_search(std::string P, std::string T, std::string alpha, std::vector<int> & outList){
```

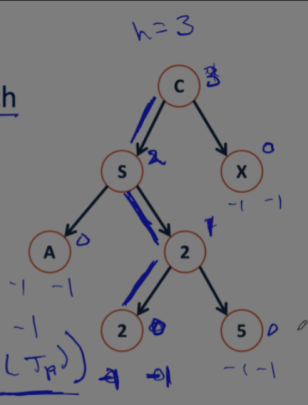But what if I was actually trying to look up 'string'?

Limitations of exact pattern matching

If I ban "bad word", what happens to "b@d w0rd"?

Approximate Pattern Matching

Approximate Pattern Matching

```
Score =  248 bits (129), Expect = 1e-63
Identities = 213/263 (80%), Gaps = 34/263 (12%)
Strand = Plus / Plus


Query: 161 atatcaccacgtcaaaggtgactccaactcca---ccactccattttgttcagataatgc 217
            ||||||||||||||||||||||||||||||| |     | |    || |||||||||||||
Sbjct: 481 atatcaccacgtcaaaggtgactccaact-tattgatagtgttttatgttcagataatgc 539


Query: 218 ccgatgatcatgtcatgcagctccaccgattgtgagaacgacagcgacttccgtcccagc 277
            |||||||    |||||||||||||||||||| || |              |||||||||||
Sbjct: 540 ccgatgactttgtcatgcagctccaccgattttg-g-----------ttccgtcccagc 586


Query: 278 c-gtgcc--aggtgctgcctcagattcaggttatgccgctcaattcgctgcgtatatcgc 334
            |   || |   | |||||||||||||||||||||||||||||||||||||| |||||||||
Sbjct: 587 caatgacgta-gtgctgcctcagattcaggttatgccgctcaattcgctgggtatatcgc 645


Query: 335 ttgctgattacgtgcagctttcccttcaggcggga-----------ccagccatccgtc 382
            |||||||||||||||||||||||||||||||||||           |||||||||||||
Sbjct: 646 ttgctgattacgtgcagctttcccttcaggcgggattcatacagcggccagccatccgtc 705


Query: 383 ctccatatc-accacgtcaaagg 404
            ||||||||| |||||||||||||
Sbjct: 706 atccatatcaaccacgtcaaagg 728
```

Approximate Pattern Matching

Input: A text *T*, a pattern *P*, and a distance *d*

Output: All positions in *T* where *P* has at most *d* mismatches or edits

P: word

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

Alignment 1: word

Alignment 2: word

~~Not a match!~~

Match!

Distance 2 match!

Distance 0 match!

Approximate Pattern Matching

What is the distance between these two strings?

X: **1 0 0 1 1**

Y: **0 0 1 1 0**

Approximate Pattern Matching

What is the distance between these two strings?

X: **1 0 0 1 1**

Y: **0 0 1 1 0**

X: **1 0 0 1 1**
      | |
Y: **0 0 1 1 0**

Hamming distance is 3!

X: **1 0 0 1 1 -**
        | | | |
Y: **- 0 0 1 1 0**

Edit distance is 2!

Approximate Pattern Matching

How can I describe the relationship between two strings?

X: **1 0 0 1 1**

Y: **0 0 1 1 0**

X: **1 0 0 1 1 -**

Y: **- 0 0 1 1 0**

Approximate Pattern Matching

A **substitution** replaces one character with another

Described as the character swap needed to *convert T* to *P*
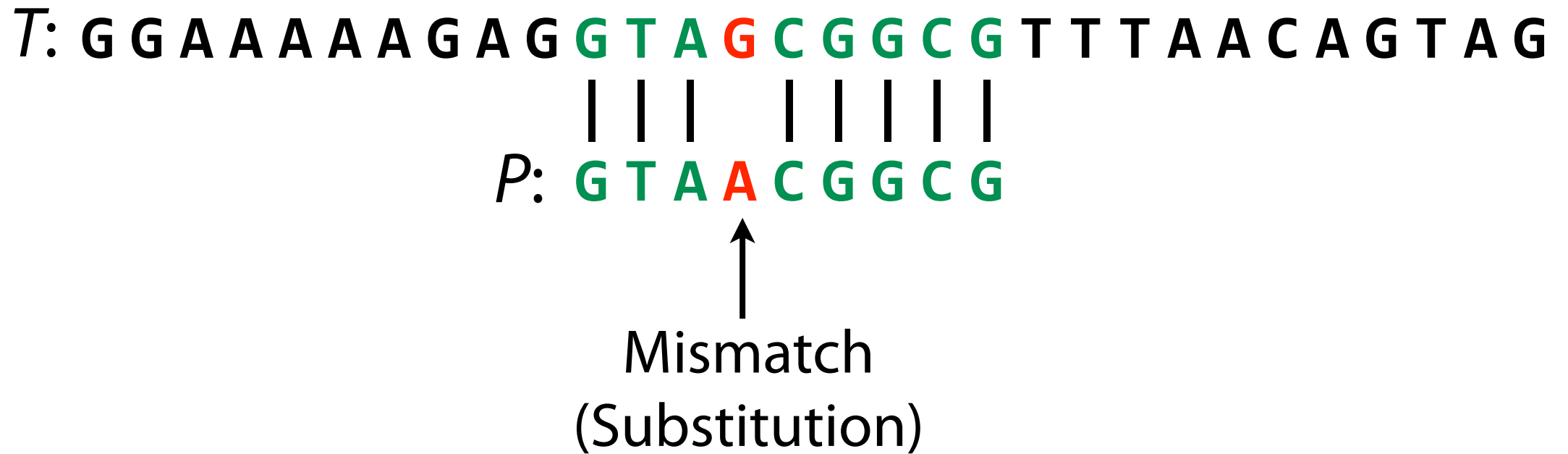
T: G G A A A A A G A G G T A G C G G C G T T T A A C A G T A G

P: G T A A C G G C G

Mismatch
(Substitution)

Hamming Distance

The minimum number of **substitutions** to turn one string into another.

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**

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 T**

Hamming Distance

The minimum number of **substitutions** to turn one string into another.

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

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 T**

Hamming Distance

The minimum number of **substitutions** to turn one string into another.

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

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 T**

Hamming distance = 6

Hamming Distance

The minimum number of **substitutions** to turn one string into another.

X: **G G C C G G C**

Y: **C C G G G G G**

X: **T A T A T A**

Y: **A T A T A T**

Hamming Distance

The minimum number of **substitutions** to turn one string into another.

X: G G C C G G C

| |

Y: C C G G G G G

Hamming distance = 5

X: T A T A T A

Y: A T A T A T

Hamming Distance

The minimum number of **substitutions** to turn one string into another.

X: **G G C C G G C**

| |

Y: **C C G G G G G**

Hamming distance = 5

X: **T A T A T A**

Y: **A T A T A T**

Hamming distance = 6

Approximate Pattern Matching

An **insertion** adds a character, shifting all other characters back

Insertion is relative! What edits *convert T to P*

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
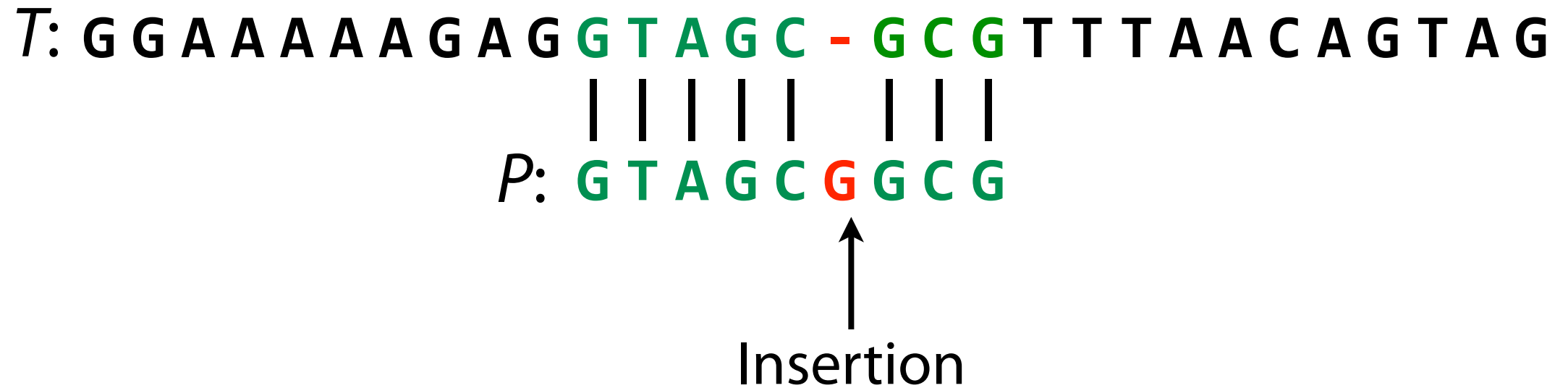
Insertion

Approximate Pattern Matching

An **deletion** removes a character, shifting all other characters forward

Deletion is relative! What edits *convert T* to *P*
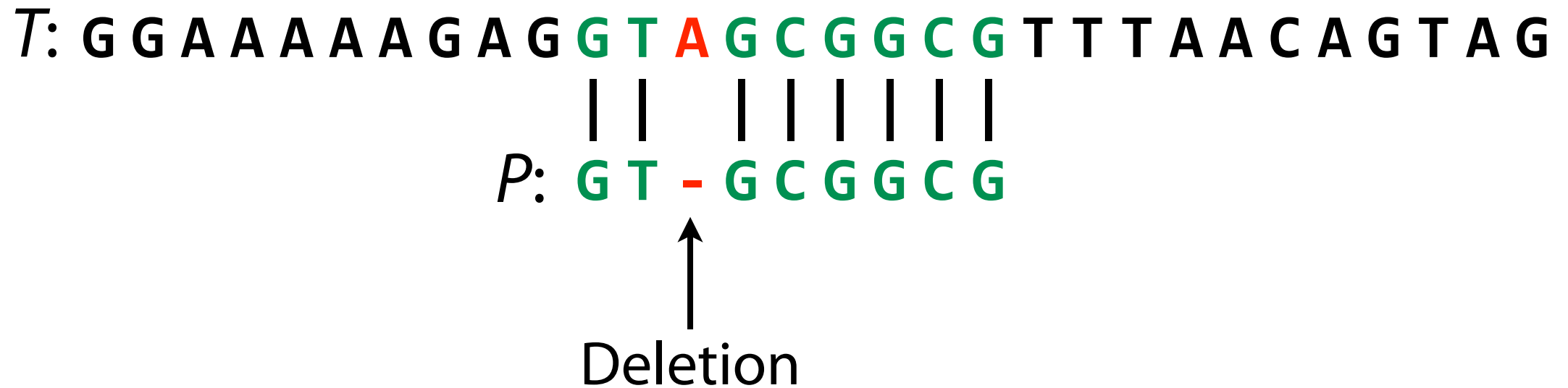
T: **G G A A A A A G A G G T A G C G G C G T T T A A C A G T A G**

P: **G T - G C G G C G**

Deletion

Edit Distance

The minimum number of substitutions, insertions, or deletions (**edits!**) needed to turn one string into another (from X to Y)!

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 T**

X: **G C G C T**

Y: **G C T**

Edit Distance

The minimum number of substitutions, insertions, or deletions (**edits!**) needed to turn one string into another (from X to Y)!

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 T**

Edit distance = 3

X: **G C G C T**

Y: **G C T**

Edit Distance

The minimum number of substitutions, insertions, or deletions (**edits!**) needed to turn one string into another (from X to Y)!

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 – C A G C **T**

Edit distance = 3

X: **G** C G C T   G C **G** C T   G **G** C C T

| | | | | | | | | |

Y: – – G C T   G C – – T   G – – C T

Edit distance = 2

Edit Distance

The minimum number of substitutions, insertions, or deletions (**edits!**) needed to turn one string into another (from X to Y)!

X: **G G C C G G C**

Y: **C C G G G G G**

X: **T A T A T A**

Y: **A T A T A T**

Edit Distance

The minimum number of substitutions, insertions, or deletions (**edits!**) needed to turn one string into another (from X to Y)!

X: G G C C G G C   G G C C G G C – –

               | | | | | | *Edit distance = 5*

Y: C C G G G G G   – – C C G G G G G

X: T A T A T A

Y: A T A T A T

Edit Distance

The minimum number of substitutions, insertions, or deletions (**edits!**) needed to turn one string into another (from X to Y)!

X: **G G C C G G C**

Y: **C C G G G G G**

X: **G G C C G G C - -**

Y: **- - C C G G G G G**

Edit distance = 5

X: **T A T A T A -**

Y: **- A T A T A T**

Edit distance = 2

Edit Distance

Carrt

Cart
Carrot
Caret

1 *deletion*

1 *insertion*

c a r t ◄──────── c a r r t ────────► c a r r o t

1 *substitution*

│
▼

c a r r y

c a r e t
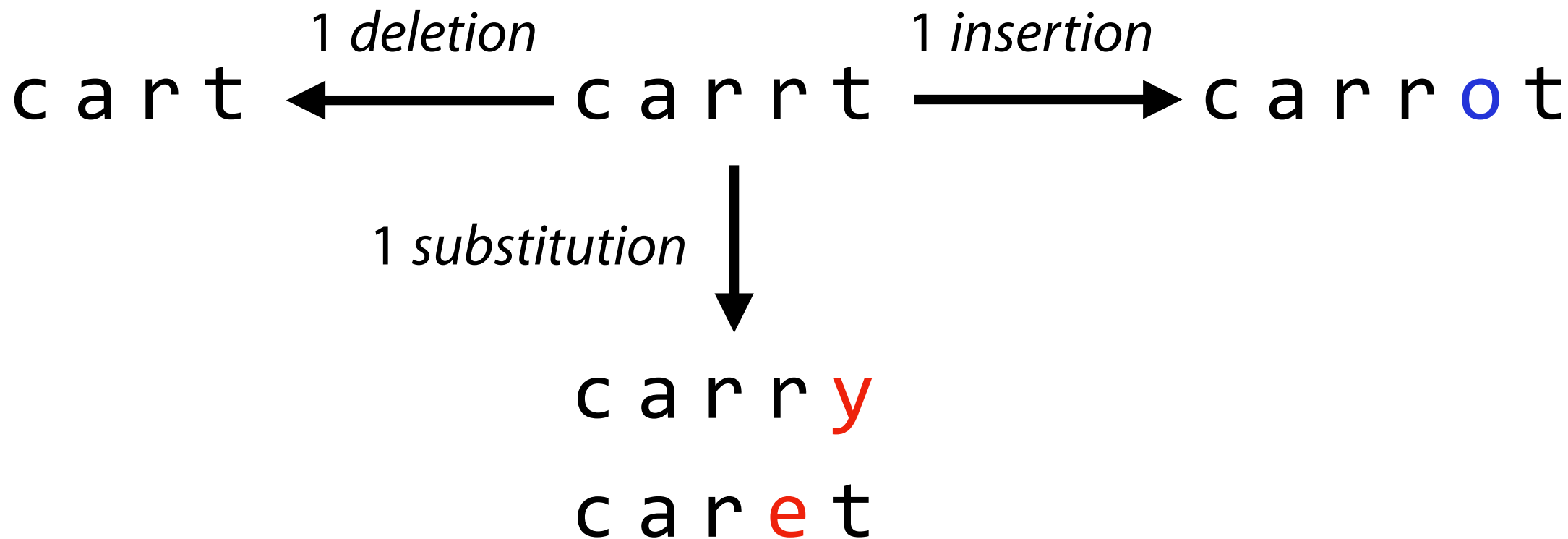
Edit Distance

```
Score =  248 bits (129), Expect = 1e-63
Identities = 213/263 (80%), Gaps = 34/263 (12%)
Strand = Plus / Plus
```

Substitution

```
Query: 161 atatcaccacgtcaaaggtgactccaactcca---ccactccattttgttcagataatgc 217
            ||||||||||||||||||||||||||||||| |     | |     || ||||||||||||
Sbjct: 481 atatcaccacgtcaaaggtgactccaact-tattgatagtgttttatgttcagataatgc 539


Query: 218 ccgatgatcatgtcatgcagctccaccgattgtgagaacgacagcgacttccgtcccagc 277
            |||||||      ||||||||||||||||||||| || |             |||||||||||
Sbjct: 540 ccgatgactttgtcatgcagctccaccgattttg-g-----------ttccgtcccagc 586
```

Deletion

```
Query: 278 c-gtgcc--aggtgctgcctcagattcaggttatgccgctcaattcgctgcgtatatcgc 334
            |   || |   |  ||||||||||||||||||||||||||||||||||||| ||||||||||
Sbjct: 587 caatgacgta-gtgctgcctcagattcaggttatgccgctcaattcgctgggtatatcgc 645


Query: 335 ttgctgattacgtgcagctttcccttcaggcggga------------ccagccatccgtc 382
            |||||||||||||||||||||||||||||||||||            ||||||||||||||
Sbjct: 646 ttgctgattacgtgcagctttcccttcaggcgggattcatacagcggccagccatccgtc 705
```

Insertion

```
Query: 383 ctccatatc-accacgtcaaagg 404
            ||||||||| ||||||||||||||
Sbjct: 706 atccatatcaaccacgtcaaagg 728
```

Approximate Pattern Matching

How can I describe the relationship between two strings?

X: **1 0 0 1 1**

Y: **0 0 1 1 0**

X: **1 0 0 1 1 -**

Y: **- 0 0 1 1 0**

Edit string: Describe the changes you would make to X to become Y

Approximate Pattern Matching

Input: A text *T*, a pattern *P*, and a distance *d*

Output: All positions in *T* where *P* has at most *d* mismatches or edits

Hamming Distance: Min number substitutions (mismatches)

Edit Distance: Min number edits (substitution, insertions, deletions)

Approximate Pattern Matching

$\Sigma = 0, 1 \qquad P = 000$

Hamming Distance 1 strings:

Edit Distance 1 strings:

Approximate Pattern Matching

$\Sigma = 0, 1 \qquad P = 000$

Hamming Distance 1 strings: 100   010   001

Edit Distance 1 strings: 100   1000   00

010   0100

001   0010

0001

0000

Approximate Pattern Matching

$P = \text{abb}$ $d = 1$

Using **Hamming** distance, what are valid approximate matches for P?

Using **edit** distance, what are valid approximate matches for P?

A) aba

B) aabb

C) bbb

D) ab

Approximate Pattern Matching

How do we find all approximate matches for a pattern in a text?

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

Approximate Pattern Matching

How do we find all approximate matches for a pattern in a text?

Can we use our efficient exact pattern matching algorithms?

	A	**C**	**G**	**T**
A	0	0	1	2
C	0	1	0	1
G	0	1	2	0
T	0	1	2	3
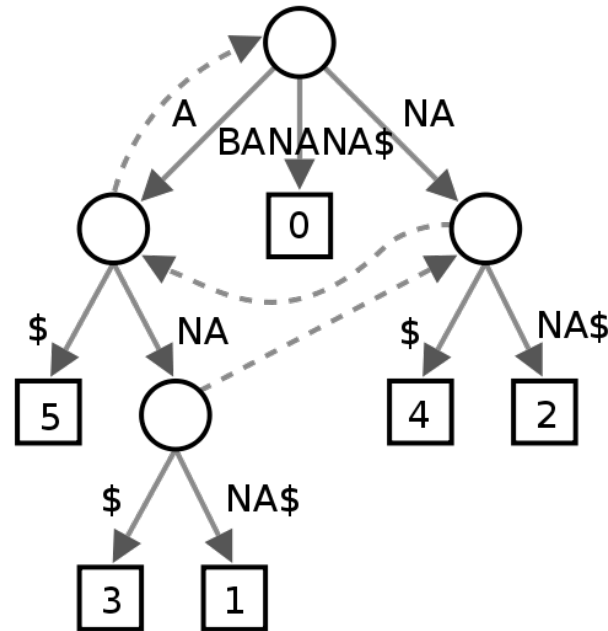
6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

$ B A N A N A
A $ B A N A N
A N A $ B A N
A N A N A $ B
B A N A N A $
N A $ B A N A
N A N A $ B A

Boyer-Moore   Suffix Tree   Suffix Array   FM Index

Approximate Pattern Matching

Can we use our efficient exact pattern matching algorithms?

For Hamming distance (mismatches), we can!

$$\overline{\qquad\qquad P \qquad\qquad}$$

101110101101101010001010

Approximate Pattern Matching

Can we use our efficient exact pattern matching algorithms?

For Hamming distance (mismatches), we can!

$$\overline{\qquad\qquad\qquad P \qquad\qquad\qquad}$$

1011101011011010001010

Approximate Pattern Matching

Can we use our efficient exact pattern matching algorithms?

For Hamming distance (mismatches), we can!

$$\underline{\ P\ }$$

101110101101101000**1010**

10111110110**11010001010**

10111010110110100**0**010

10111010111110100001010

1011101011011**11**0001010

Approximate Pattern Matching

Can we use our efficient exact pattern matching algorithms?

For Hamming distance (mismatches), we can!

$$P$$

| U | V |
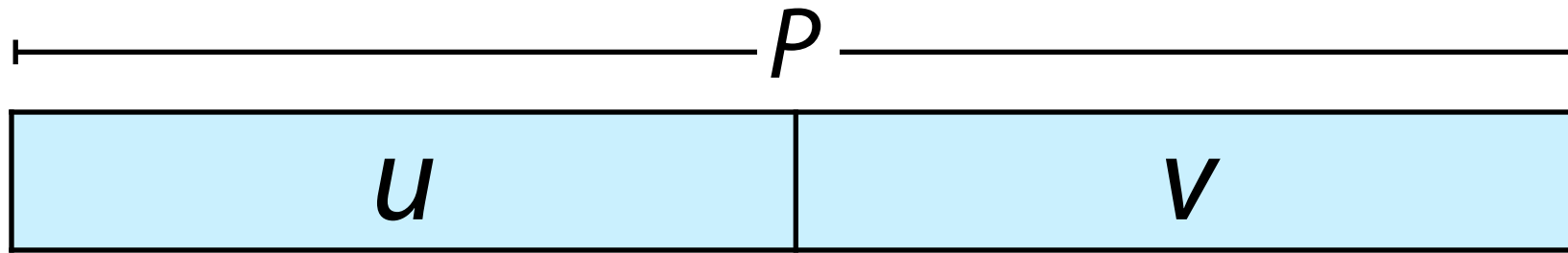
If P occurs in T with 1 mismatch, then u or v has no mismatch

We can search for u and v in T as a proxy for P!

Approximate Pattern Matching
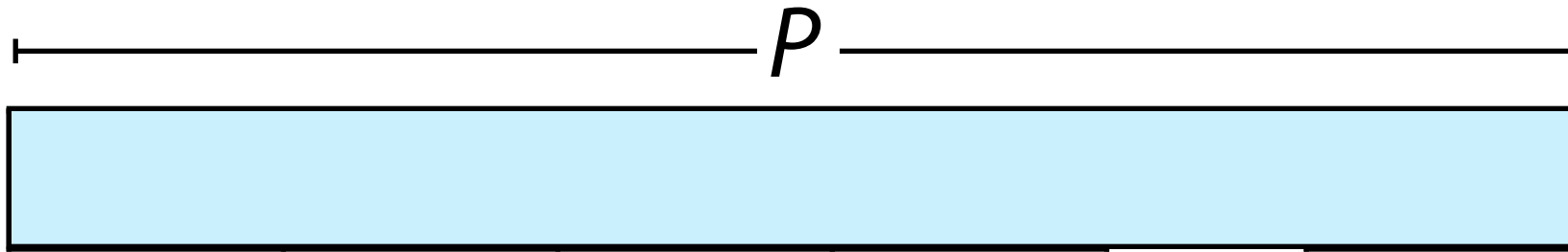
If *P* occurs in *T* with up to *k* mismatches…

P

Approximate Pattern Matching

If *P* occurs in *T* with up to *k* mismatches, then if we split P into *k+1* partitions, at least one of p_0, p_1, ..., p_k must appear with 0 mismatches.
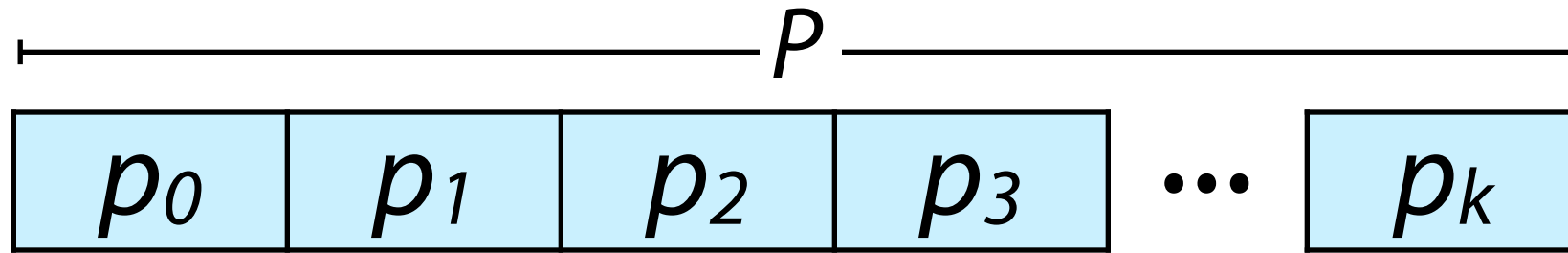
Approximate Pattern Matching

If *P* occurs in *T* with up to *k* mismatches, then if we split P into *k+1* partitions, at least one of p_0, p_1, ..., p_k must appear with 0 mismatches.
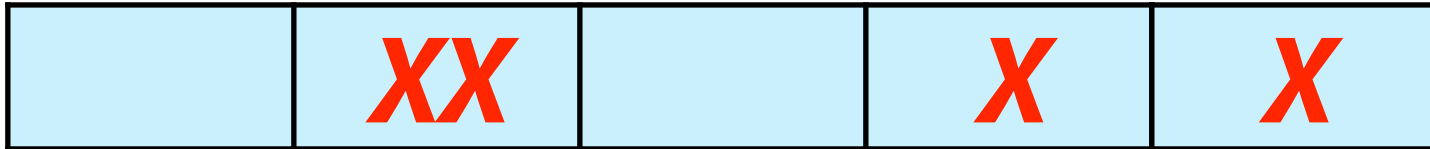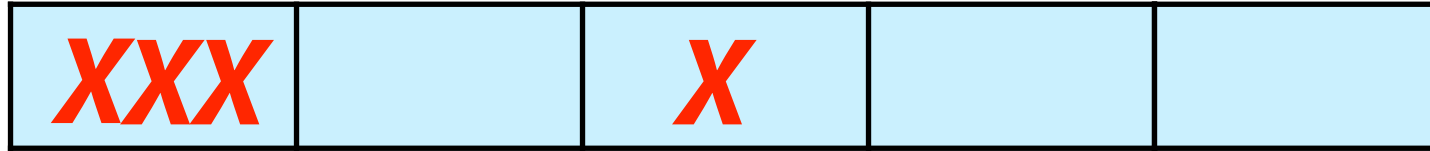
5 partitions
4 mismatches (*X*)

Approximate Pattern Matching

Pigeonhole principle: A direct relationship between containers and objects from either perspective below.

k+1 pigeons, *k* holes?

At least one hole has two pigeons!

k pigeons, *k+1* holes?

At least one hole is empty!

Approximate Pattern Matching

Pigeonhole principle lets us use exact matching algorithms:

P: `word`

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

Approximate Pattern Matching

Pigeonhole principle lets us use exact matching algorithms:

P: `word`

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

u: `wo` `wo` `wo`

v: `rd` `rd`

1) Given *k* allowed mismatches, break the pattern up into *k+1* partitions

What do we do with these partial matches?

Approximate Pattern Matching

Counting mismatches requires verifying non-matching partitions
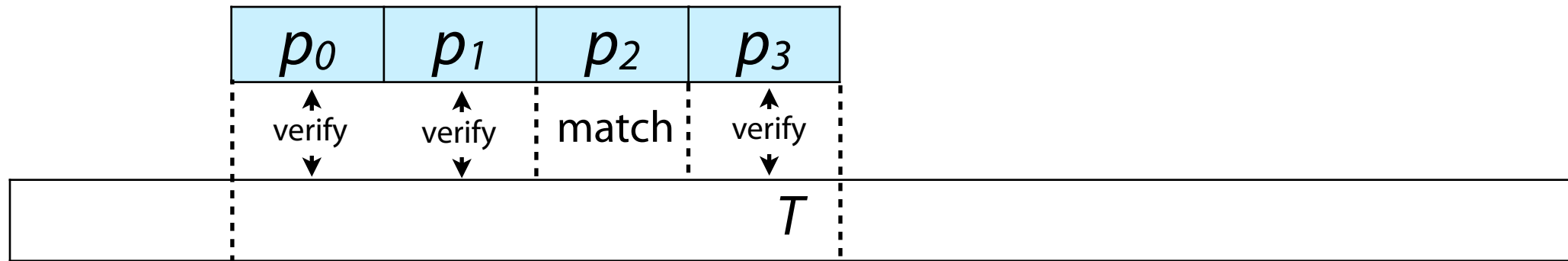
Approximate Pattern Matching

This is known as the **seed and extend** heuristic

T: **There would have been a time for such a word**
     word   **word**
                                                        word
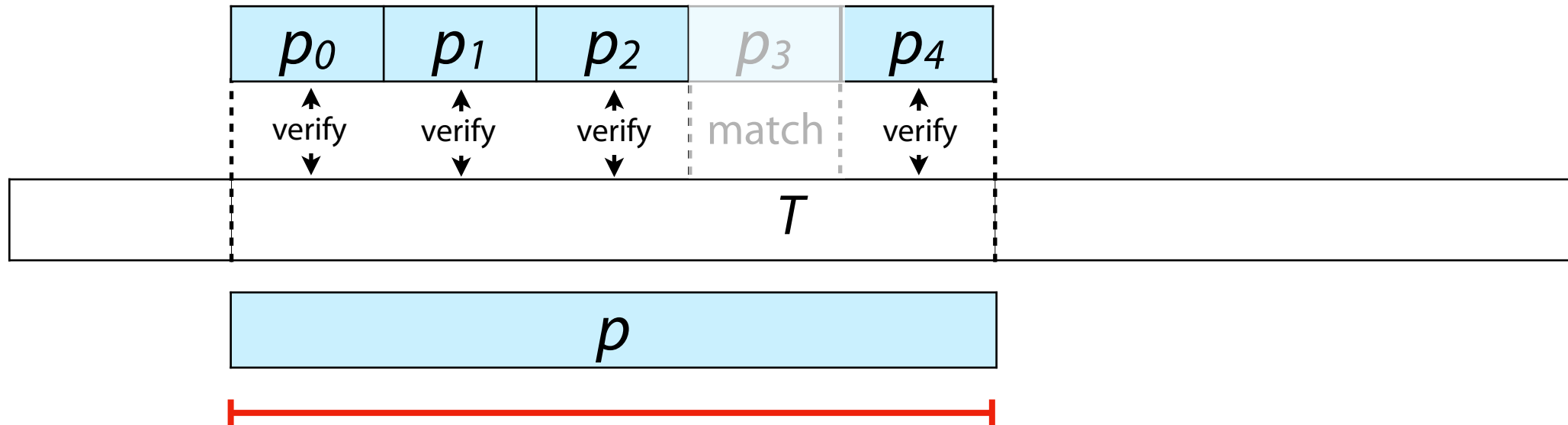
Only consider mismatches while verifying a seed hit

Seed and Extend Approximate Pattern Matching

P: `word`

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

u: `wo` `word` `word`

v: `rd` `word`

1) Given *k* allowed mismatches, break the pattern up into *k+1* partitions

2) For every partial exact match, **inspect** the full alignment

   (Count mismatches in the remaining characters in the alignment)

3) Return all matches (but don't duplicate!)

Approximate Pattern Matching

Find all edit distance 2 approximate matches

P: **CATS**

T: **THE CART WAS CARRIED BY THE CATS**

Approximate Pattern Matching

Find all edit distance 2 approximate matches

P: **CATS**

T: **THE CART WAS CARRIED BY THE CATS**

C THE **C**A**R**T WAS **C**A**RR**IED BY THE **CATS**

A THE **CART** WAS **C**A**RR**IED BY THE **CATS**

T **T**HE **C**A**R**T_WAS CARRIED B**Y_T**HE **CATS**

S THE CART**_WA**S CARRIED BY THE **CAT**S

Approximate Pattern Matching

As a **heuristic**, seed and extend reduces the overall search space

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

Consider the likelihood of seeing 'wo' or 'rd' **by chance**:

$$256 \text{ characters}: \quad \frac{1}{256}^{2} = 0.000015$$

Approximate Pattern Matching in Genomics

Partition Seed: Length ~40

CTCAAACTCCTGACCTTTGGTGATCCACCCGCCTAGGCCTTC

T: Length 3 billion

Likelihood of random seed string:

$$\frac{1}{4}^{40} = 8.27e - 25$$

# of times seed will occur by chance in T:

Likelihood * (~ length)

$$2.48e - 15$$

Approximate Pattern Matching

"Seed and extend" approach to pattern matching

$$\overset{\displaystyle P}{\boxed{p_0} \; \boxed{p_1} \; \boxed{p_2} \; \boxed{p_3} \quad \cdots \quad \boxed{p_k}}$$
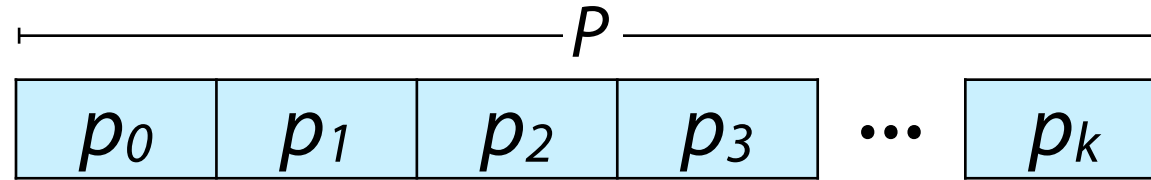
Pros: **Cons:**

Approximate Pattern Matching

"Seed and extend" approach to pattern matching

$$\overline{\; P \;}$$

| p_0 | p_1 | p_2 | p_3 | ... | p_k |
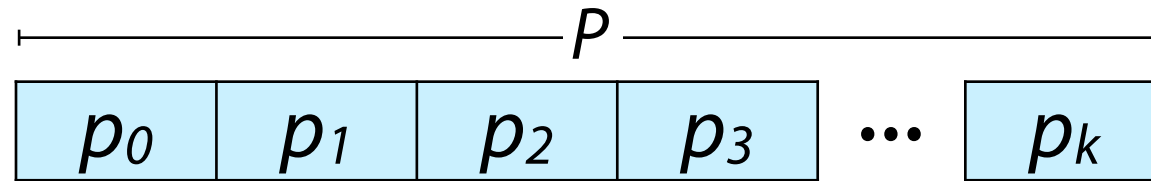
Pros:

Reuse exact matching algs

Works for Hamming and edit distance*

As a heuristic, reduces search space

Cons:

Slow for large k

small partitions matching many times by chance

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

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

Assignment 10: a_pigeon

Learning Objective:

Preprocess text into kmers and a hash table

Use pigeonhole principle to perform approximate matching

Consider: Do the partitions need to be contiguous runs of characters? Do they need to all be the same length?

`kmerMap text_to_kmer_map(string & T, int k)`

Input:

`string & T`: The input text — can be very large this week!

`int k`: The fixed size for each kmer (substring)

Output:

`kmerMap`: `unordered_map<string, vector<int>>`

T: **aaaaabbbbb**   *k = 4*

kmerMap text_to_kmer_map(string & T, int k)

Input:

   string & T: The input text — can be very large this week!

   int k: The fixed size for each kmer (substring)

Output:

   kmerMap: unordered_map<string, vector<int>>

T: **aaaaabbbbb**   *k = 4*   **Output**

  aaaa bbbb   **aaaa : {0, 1}**

   aaaa bbbb   **aaab : {2}**

    aaab bbbb   **aabb : {3}**

     aabb   **abbb : {4}**

      abbb   **bbbb : {5, 6, 7}**

```
vector<Seed> partitionPattern(string P, int np)
```

Input:

    `string P:` The input pattern — can be large this week!

    `int np:` The number of non-overlapping partitions to split P

Output:

    `vector<Seed>` : Vector of partitioned strings and their index

    `typedef std::pair<std::string, int> Seed;`

P: **ABCDEFGH**   *np = 2*

`vector<Seed> partitionPattern(string P, int np)`

Input:

`string P`: The input pattern — can be large this week!

`int np`: The number of non-overlapping partitions to split P

Output:

`vector<Seed>`: Vector of partitioned strings and their index

`typedef std::pair<std::string, int> Seed;`

P: **ABCDEFGH**   *np = 2*   `{ {ABCD, 0}, {EFGH, 4} }`

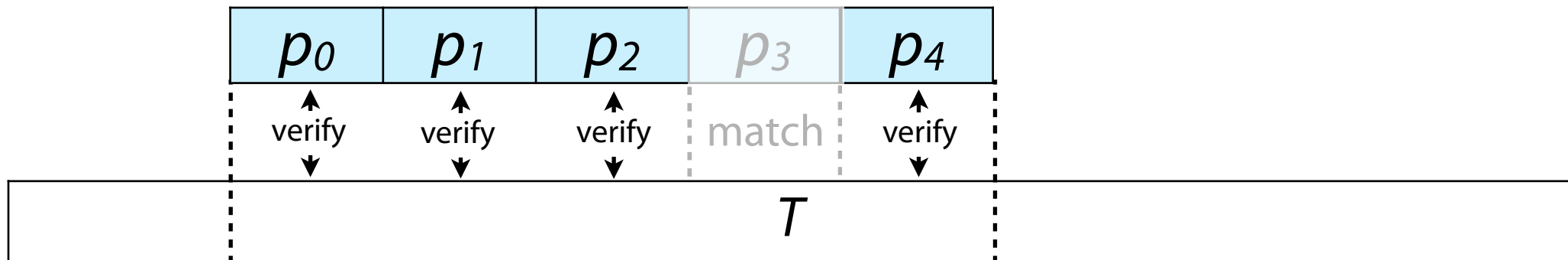# vector<int> approximate_search(fname, P, mm)

Input:

string fname: The file storing the text T

string P: The pattern text

int mm: The number of allowed mismatches

Output:

vector<int>: The index positions in T of all approximate matches

Bonus Slides

FM Index w/ mismatches

Start with shortest suffix, then match successively longer suffixes

Keep track of mismatches for each suffix

$P = $ **ab$a$**

	F						L
	$	a	b	a	a	b	a_0
	a_0	$	a	b	a	a	b
	a_1	a	b	a	$	a	b
	a_2	b	a	$	a	b	a_1
	a_3	b	a	a	b	a	$
	b	a	$	a	b	a	a_2
	b	a	a	b	a	$	a_3

Easy to find all the rows beginning with **a**

FM Index w/ mismatches

Start with shortest suffix, then match successively longer suffixes

Keep track of mismatches for each suffix

$P = $ **ab$a$**

But we count everything else as a potentially valid mismatch

	F						L
	$	a	b	a	a	b	**a_0**
	a_0	$	a	b	a	a	**b**
	a_1	a	b	a	$	a	**b**
	a_2	b	a	$	a	b	**a_1**
	a_3	b	a	a	b	a	**$**
	b	a	$	a	b	a	**a_2**
	b	a	a	b	a	$	**a_3**

FM Index w/ mismatches

We have rows beginning with **a**, now we want rows beginning with **ba**
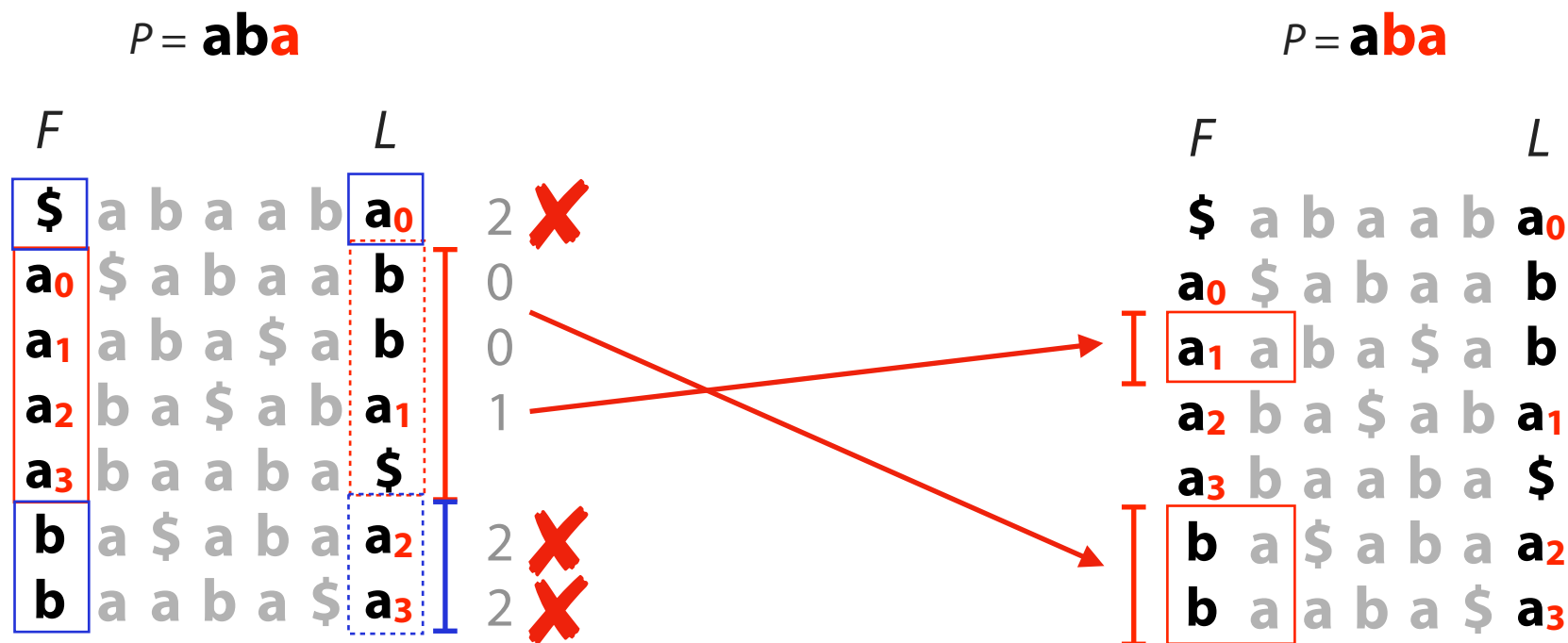
No longer have just one search range!

FM Index w/ mismatches

We have rows beginning with **ba**, now we seek rows beginning with **aba**

$P =$ **a****ba**

F						L	
$	a	b	a	a	b	**a**$_0$	
a$_0$	$	a	b	a	a	**b**	
a$_1$	a	b	a	$	a	**b**	2 ✗
a$_2$	b	a	$	a	b	**a**$_1$	
a$_3$	b	a	a	b	a	**$**	
b	a	$	a	b	a	**a**$_2$	0
b	a	a	b	a	$	**a**$_3$	0

$P =$ **aba**

F						L
$	a	b	a	a	b	**a**$_0$
a$_0$	$	a	b	a	a	**b**
a$_1$	a	b	a	$	a	**b**
a$_2$	b	a	$	a	b	**a**$_1$
a$_3$	b	a	a	b	a	**$**
b	a	$	a	b	a	**a**$_2$
b	a	a	b	a	$	**a**$_3$

Only works for Hamming Distance (mismatches)!