String Algorithms and Data Structures
Approximate Pattern Matching

CS 199-225
Brad Solomon

November 7, 2022

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…

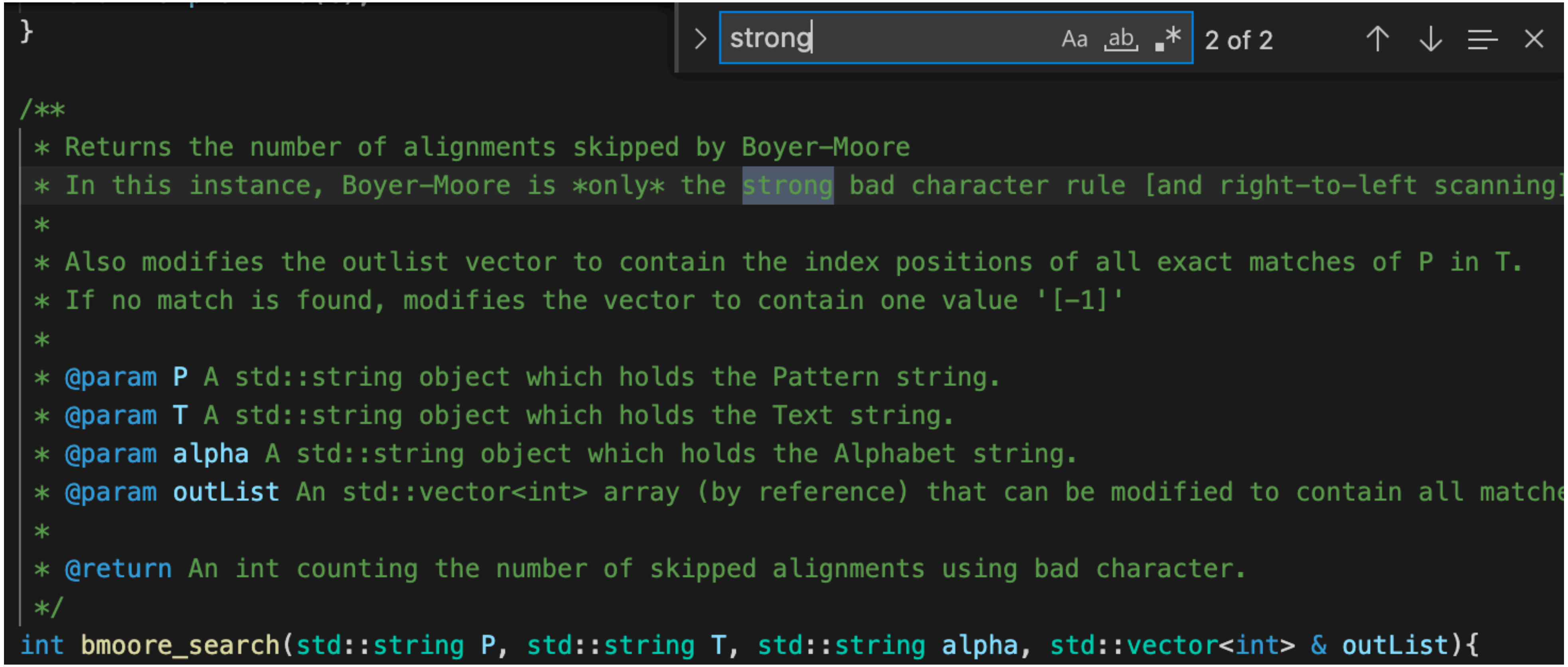
- … With mismatches
- … With edits

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

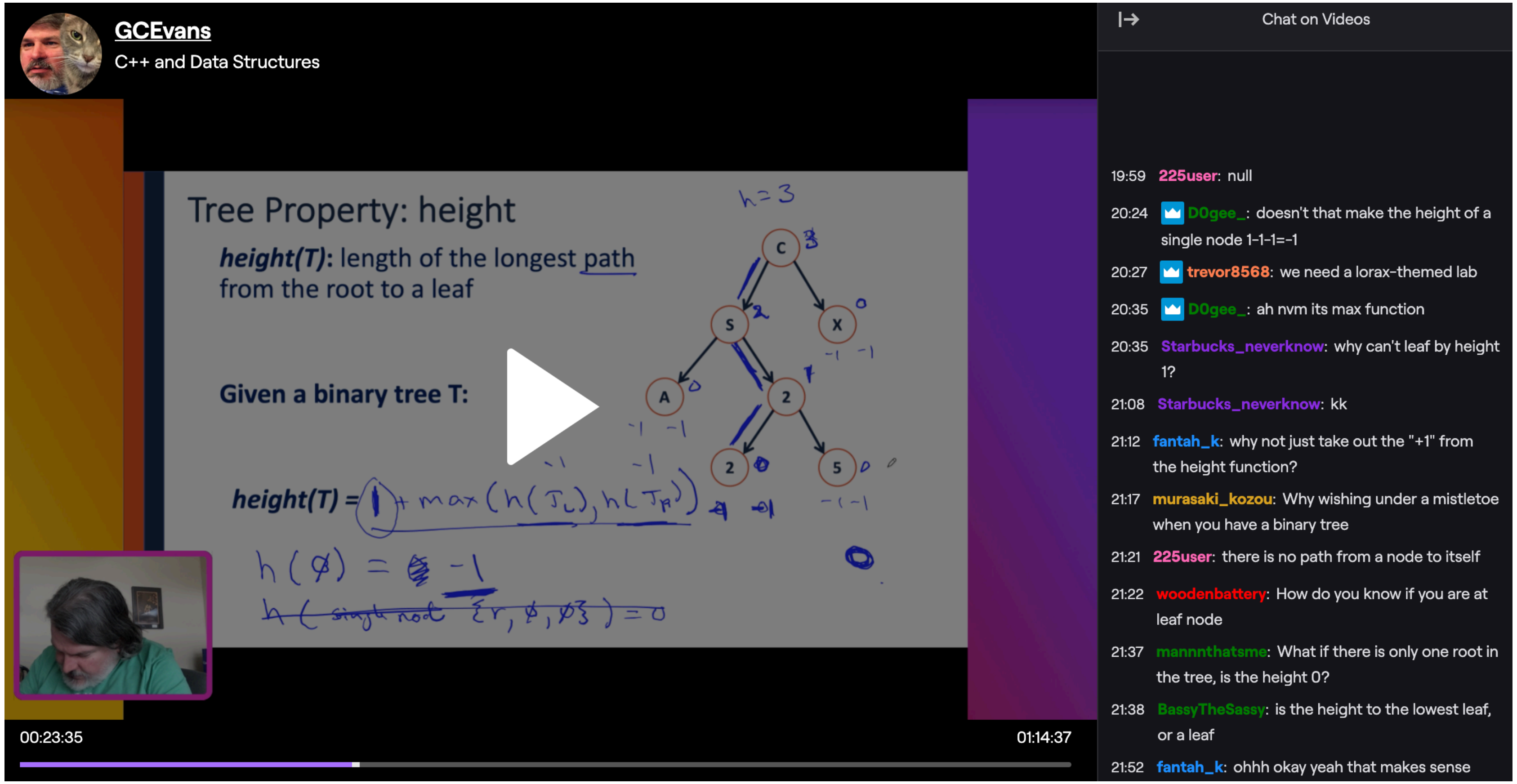
Limitations of exact pattern matching



```
}

/**
 * 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



Tree Property: height

height(T): length of the longest path from the root to a leaf

Given a binary tree T:

height(T) = 1 + max(h(T_L), h(T_R))

h(∅) = -1

19:59 225user: null

20:24 D0gee_: doesn't that make the height of a single node 1-1-1=-1

20:27 trevor8568: we need a lorax-themed lab

20:35 D0gee_: ah nvm its max function

20:35 Starbucks_neverknow: why can't leaf by height 1?

21:08 Starbucks_neverknow: kk

21:12 fantah_k: why not just take out the "+1" from the height function?

21:17 murasaki_kozou: Why wishing under a mistletoe when you have a binary tree

21:21 225user: there is no path from a node to itself

21:22 woodenbattery: How do you know if you are at leaf node

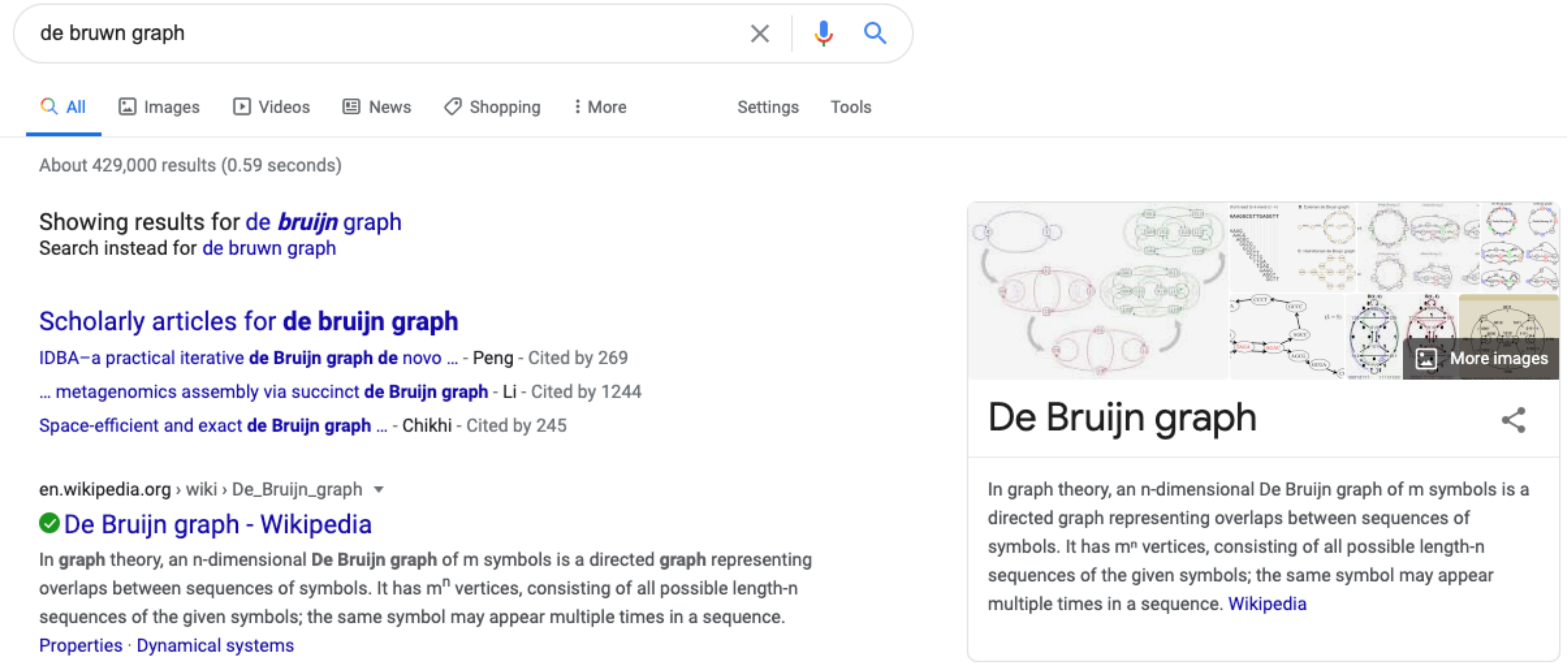
21:37 mannnthatsme: What if there is only one root in the tree, is the height 0?

21:38 BassyTheSassy: is the height to the lowest leaf, or a leaf

21:52 fantah_k: ohhh okay yeah that makes sense

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

Approximate Pattern Matching



de bruwn graph

About 429,000 results (0.59 seconds)

Showing results for de ***bruijn*** graph
Search instead for de bruwn graph

Scholarly articles for **de bruijn graph**

IDBA–a practical iterative **de Bruijn graph de** novo ... - Peng - Cited by 269

... metagenomics assembly via succinct **de Bruijn graph** - Li - Cited by 1244

Space-efficient and exact **de Bruijn graph** ... - Chikhi - Cited by 245

en.wikipedia.org › wiki › De_Bruijn_graph

De Bruijn graph - Wikipedia

In **graph** theory, an n-dimensional **De Bruijn graph** of m symbols is a directed **graph** representing overlaps between sequences of symbols. It has m^n vertices, consisting of all possible length-n sequences of the given symbols; the same symbol may appear multiple times in a sequence.

Properties · Dynamical systems

De Bruijn graph

In graph theory, an n-dimensional De Bruijn graph of m symbols is a directed graph representing overlaps between sequences of symbols. It has m^n vertices, consisting of all possible length-n sequences of the given symbols; the same symbol may appear multiple times in a sequence. Wikipedia

Approximate Pattern Matching

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

Query: 161 atatcaccacgtcaaaggtgactccaactcca---ccactccatttgttcagataatgc 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!~~

Distance 2 match!

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
```

```
X: 1 0 0 1 1
     |   |
Y: 0 0 1 1 0
```

Hamming distance is 3!

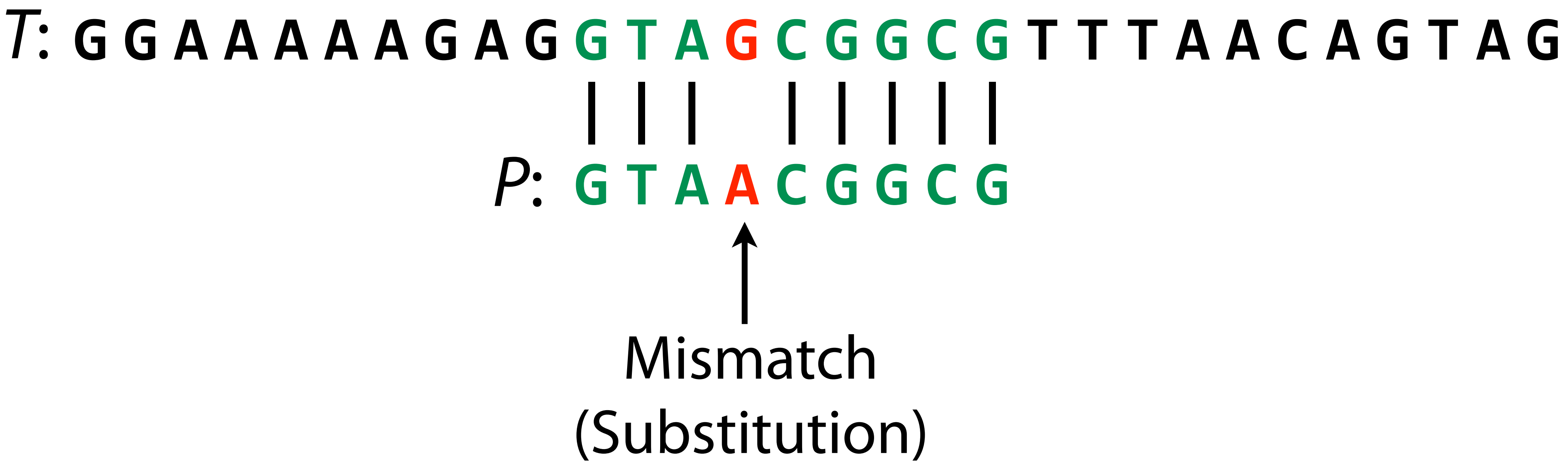
```
X: 1 0 0 1 1 -
     | | | |
Y: - 0 0 1 1 0
```

Edit distance is 2!

Approximate Pattern Matching

A **substitution** replaces one character with another

Described as the character swap needed to *convert T* to *P*



T: **GGAAAAAGAGGTAGCGGCGTTTAACAGTAG**

P: **GTAACGGCG**

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
```

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
```

Hamming distance = 5

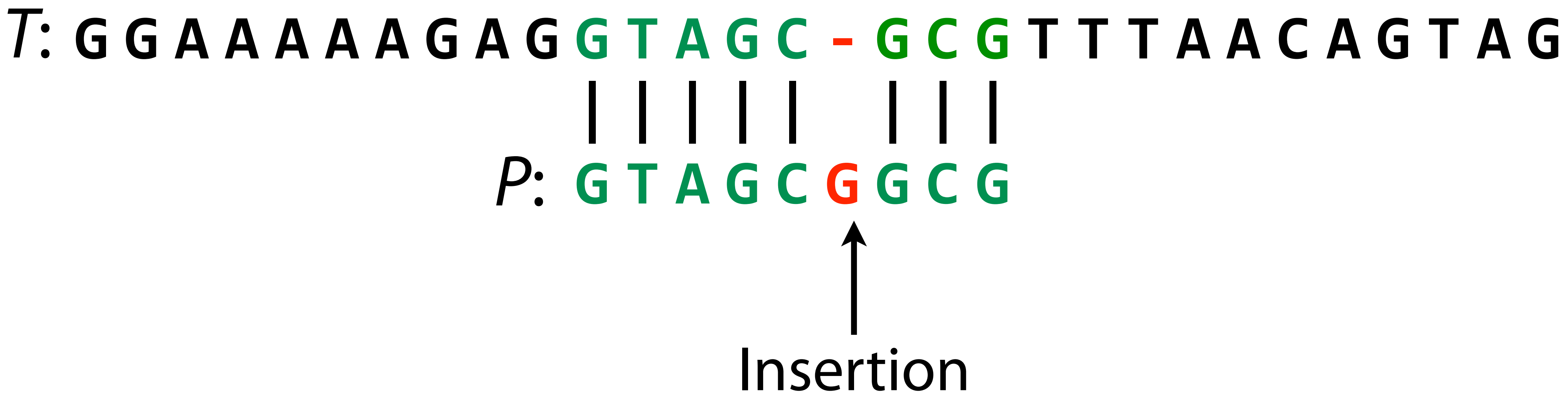
```
X:  T A T A T A

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: GGAAAAAGAGGTAGC-GCGTTTAACAGTAG
             ||||| |||
P:           GTAGCGGCG
                  ↑
              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
```

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)!

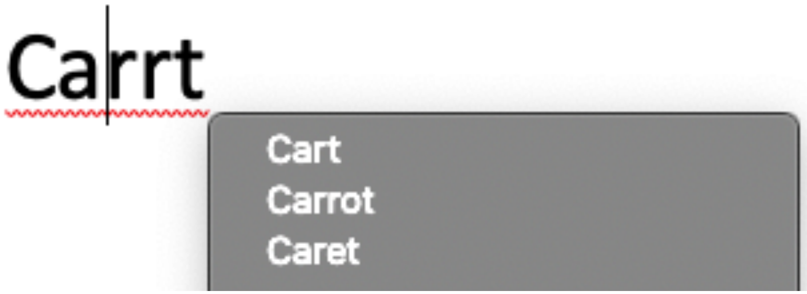
```
X: G G C C G G C          G G C C G G C - -
           | |                | | | |
Y: C C G G G G G          - - C C G G G G G
```

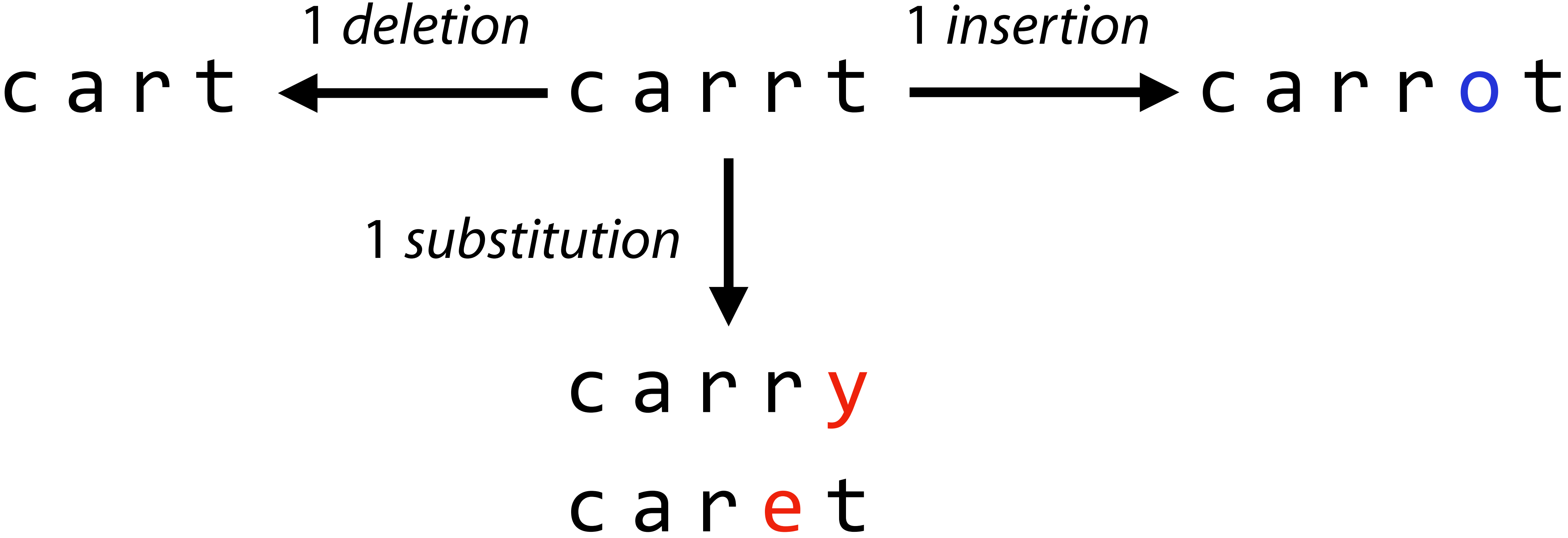
Edit distance = 5

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

Edit distance = 2

Edit Distance





cart ← *1 deletion* — carrt — *1 insertion* → carrot

carrt ↓ *1 substitution*

carry

caret

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

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` $P =$ `000`

Hamming Distance 1 strings:

Edit Distance 1 strings:

Approximate Pattern Matching

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

Hamming Distance 1 strings: 100 010 001

Edit Distance 1 strings:

100
010
001

1000
0100
0010
0001
0000

00

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?

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
```

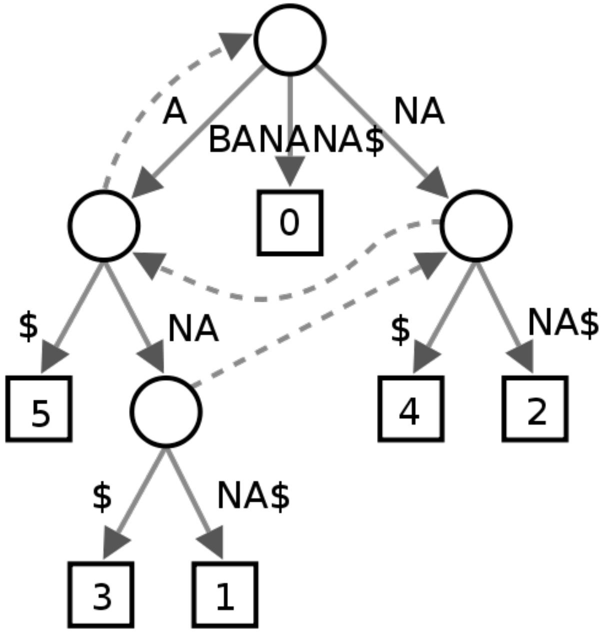
Approximate Pattern Matching

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

Boyer-Moore



Suffix Tree

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

Suffix Array

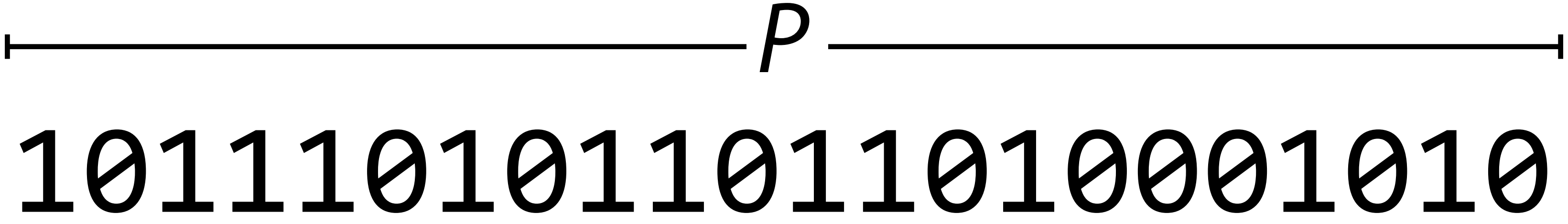
$ BANAN **A**
A $BANA **N**
A NA$BA **N**
A NANA$ **B**
B ANANA **$**
N A$BAN **A**
N ANA$B **A**

FM Index

Approximate Pattern Matching

Can we use our efficient exact pattern matching algorithms?

For Hamming distance (mismatches), we can!



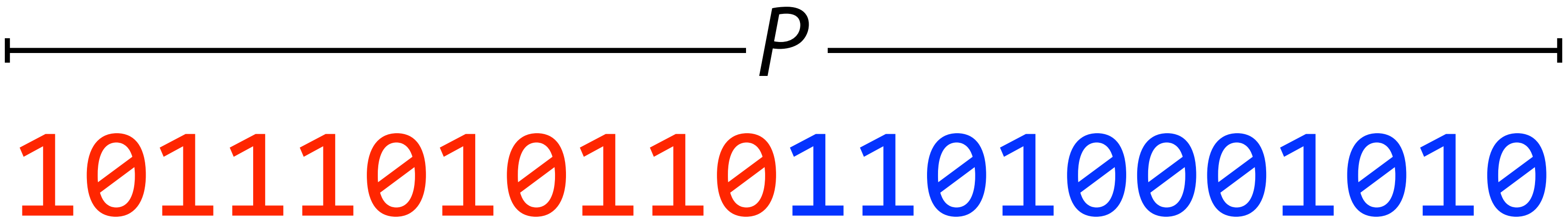
P

`1011101011011010001010`

Approximate Pattern Matching

Can we use our efficient exact pattern matching algorithms?

For Hamming distance (mismatches), we can!

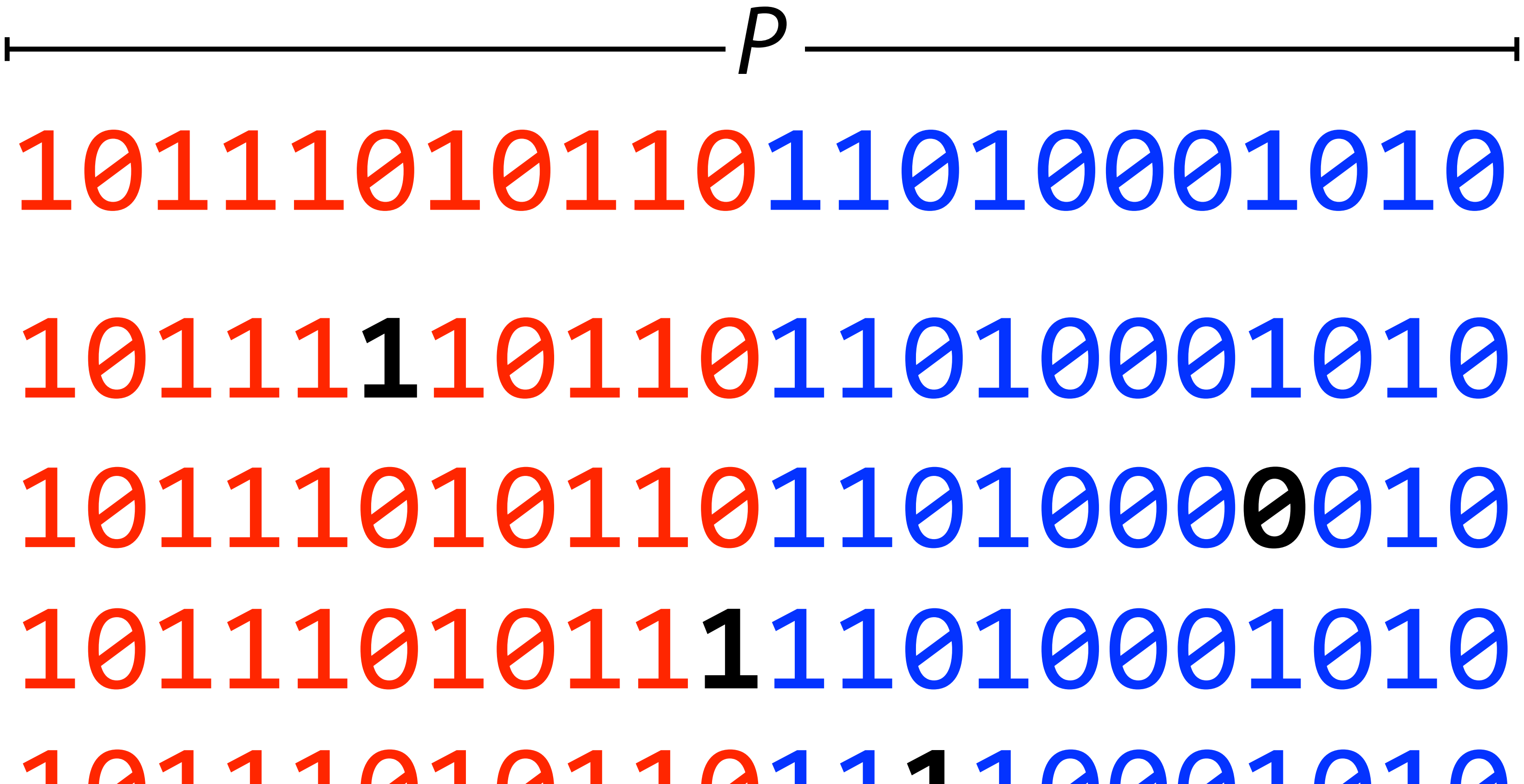


10111010110 11010001010

Approximate Pattern Matching

Can we use our efficient exact pattern matching algorithms?

For Hamming distance (mismatches), we can!



1011101011011010001010

10111**1**1011011010001010

1011101011011010000**0**010

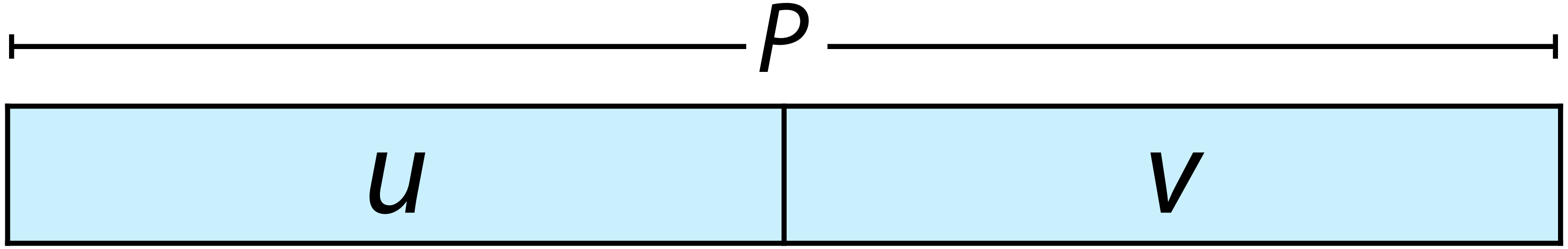
1011101011**1**11010001010

1011101011011**1**10001010

Approximate Pattern Matching

Can we use our efficient exact pattern matching algorithms?

For Hamming distance (mismatches), we can!



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

Can we use our efficient exact pattern matching algorithms?

```
P: word
T: There would have been a time for such a word
u: wo    wo                                wo
v: rd                                        rd
```

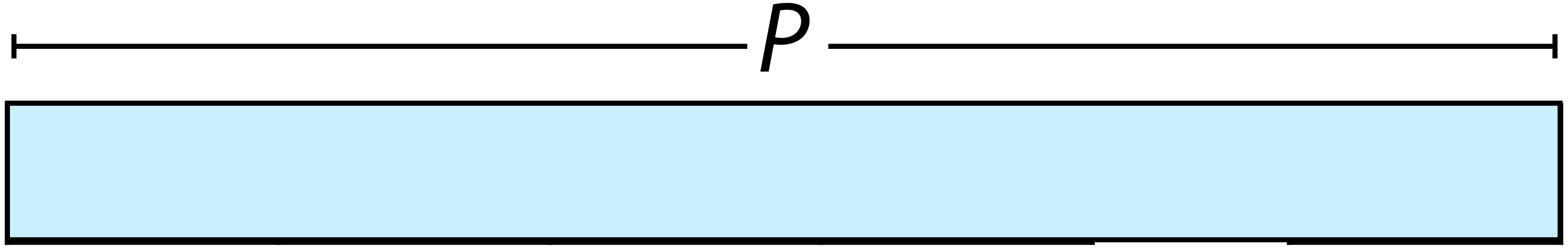
1) Search for the two half-patterns of *P*

2) Compute the number of mismatches for each half-match!

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

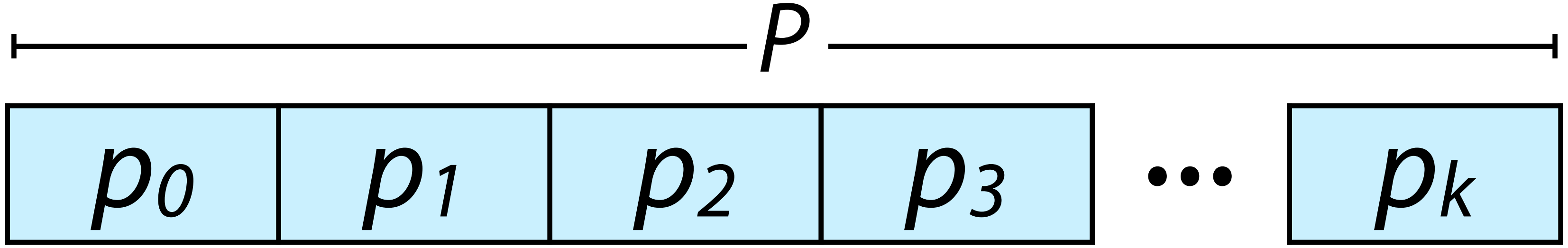
Approximate Pattern Matching

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



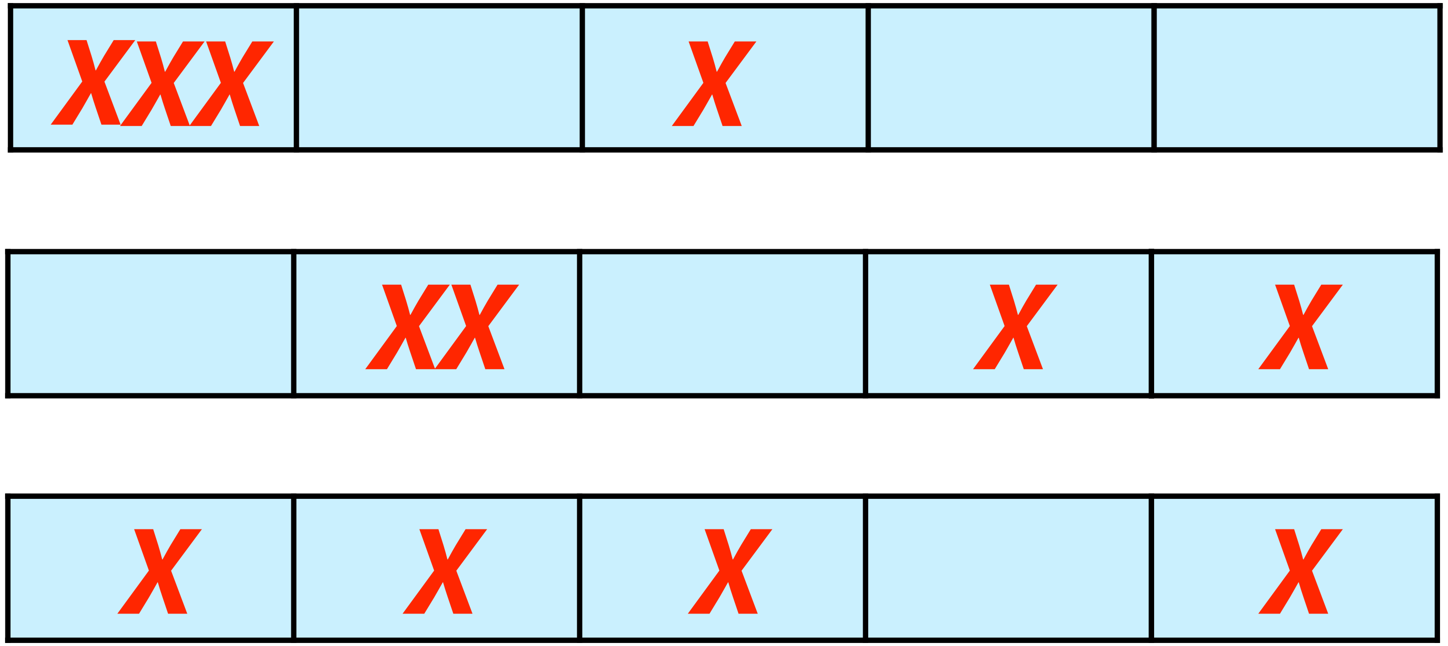
Approximate Pattern Matching

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.



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

1) Pigeonhole principle: we will have at least one exact match!

k+1 pigeons, *k* holes?

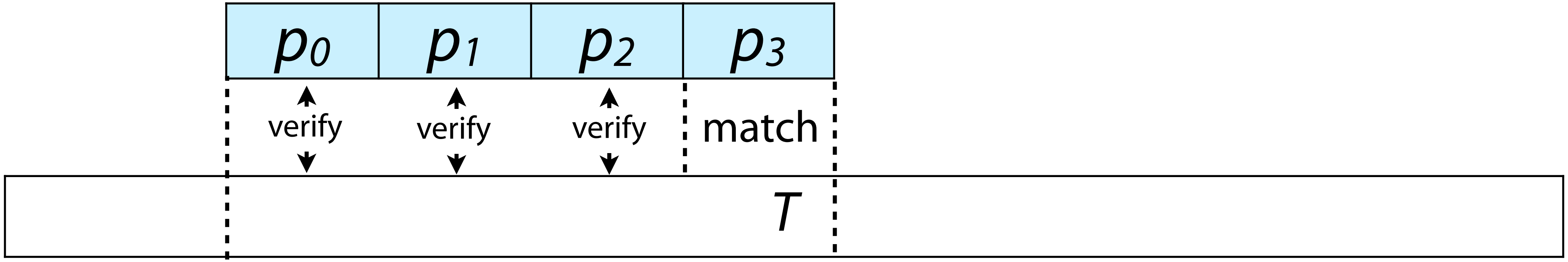
At least one hole has two pigeons!

k pigeons, *k+1* holes?

At least one hole is empty!

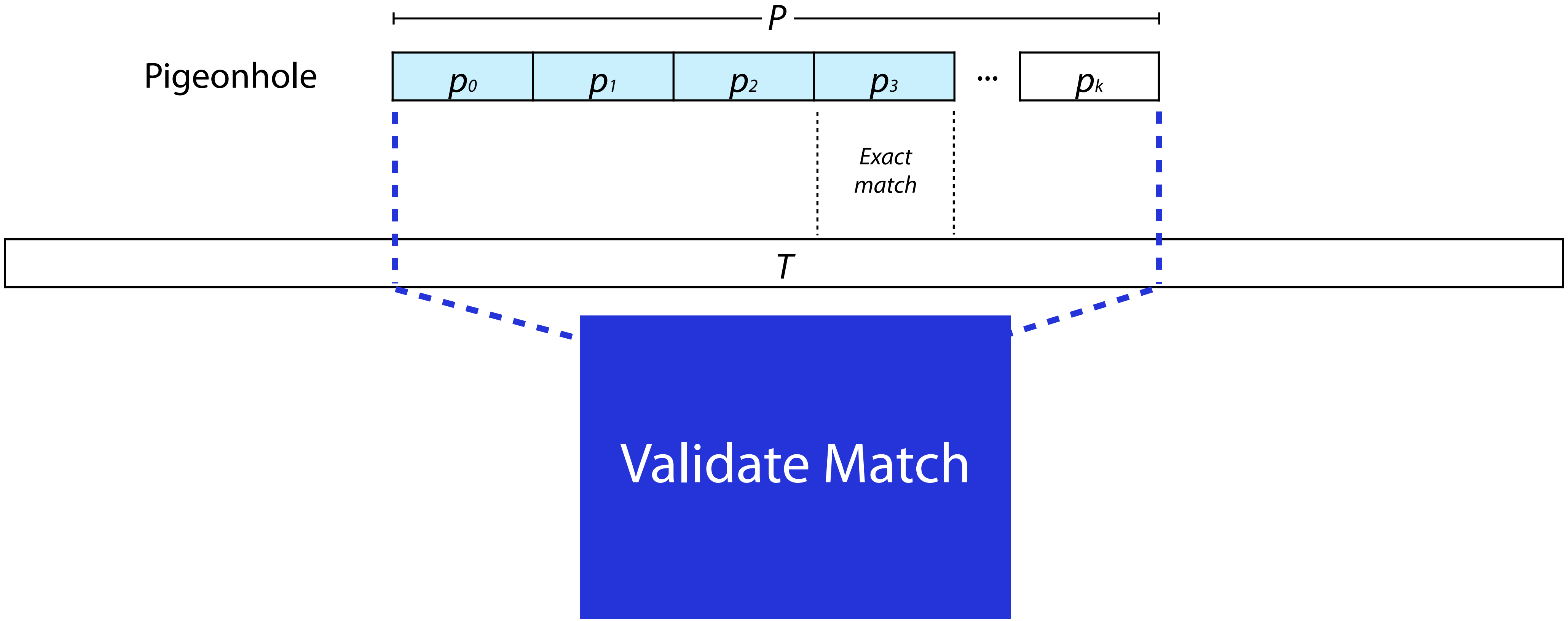
Approximate Pattern Matching

2) Verifying a match is O(P)



Seed and Extend

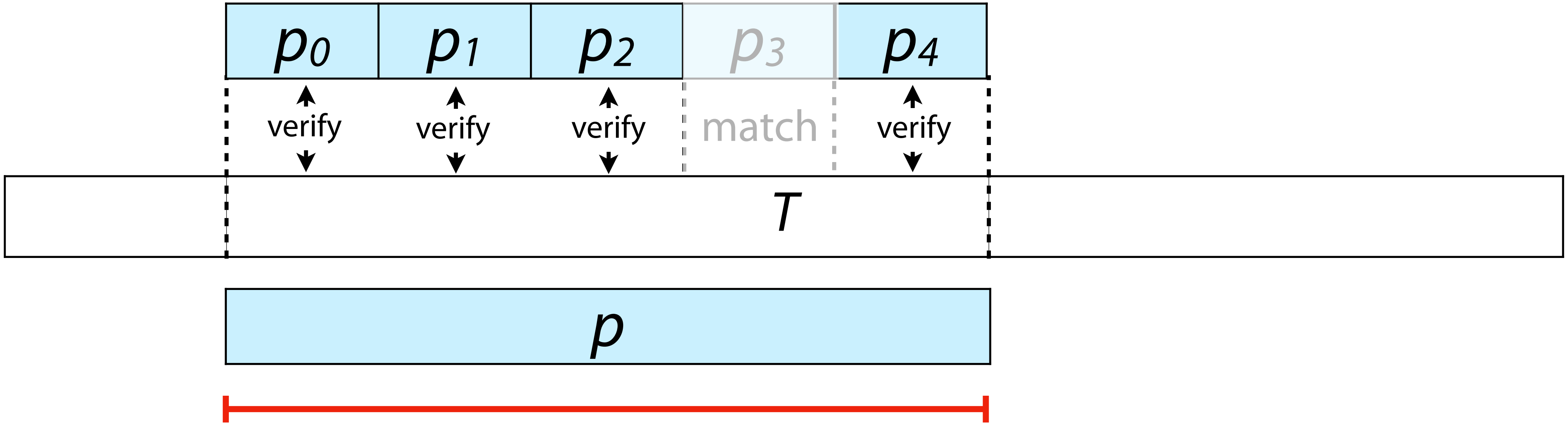
Using an exact-match ‘seed’ and ‘extending’ the match is efficient!



Seed and Extend

Exact match of ***seeds*** reduce the search space

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

       word
```



Only consider mismatches while verifying a seed hit

Seed and Extend

Exact match of ***seeds*** reduce the search space

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

Consider the likelihood of seeing 'wo' or 'rd' by chance (assuming random string):

256 characters : $\left(\frac{1}{256}\right)^2 = 0.000015$

Approximate Pattern Matching in Genomics

Partition Seed: Length ~40

`CTCAAACTCCTGACCTTTGGTGATCCACCCGCCTAGGCCTTC`

T: Length 3 billion

```
GATCACAGGTCTATCACCCTATTAACCACTCACGGGAGCTCTCCATGCATTTGGTATTTT
CGTCTGGGGGGTATGCACGCGATAGCATTGCGAGACGCTGGAGCCGGAGCACCCTATGTC
GCAGTATCTGTCTTTGATTCCTGCCTCATCCTATTATTTATCGCACCTACGTTCAATATT
ACAGGCGAACATACTTACTAAAGTGTGTTAAT[illegible]TAATGC[illegible]TAGGACATAATAATA
ACAATTGAATGTCTGCACAGCCACTTTCC[illegible]ACAGACATCATAAC[illegible]AAATTTCCACCA
AACCCCCCCTCCCCCGCTTCTGGCCACA[illegible]CACTTAAACACATCTCTGC[illegible]AACCCCAAAA
ACAAAGAACCCTAACACCAGCCTAACC[illegible]AGATTTCAAATTTTATCTTTTG[illegible]GGTATGCAC
TTTTAACAGTCACCCCCCAACTAAC[illegible]CATTATTTTCCCCTCCCACTCCCA[illegible]CTACTAAT
CTCATCAATACAACCCCCGCCCATC[illegible]TACCCAGCACACACACACCGCTGC[illegible]ACCCCATA
CCCCGAACCAACCAAACCCCAAAGA[illegible]CACCCCCCACAGTTTATGTAGCTTACC[illegible]CCTCAAA
GCAATACACTGACCCGCTCAAACT[illegible]CTGGATTTTGGATCCACCCAGCGCCTT[illegible]GCCTAAA
CTAGCCTTTCTATTAGCTCTTAGTA[illegible]GATTACACATGCAAGCATCCCCGTTC[illegible]AGTGAGT
TCACCCTCTAAATCACCACGATCAA[illegible]AGGAACAAGCATCAAGCACGCAGCA[illegible]TGCAGCTC
AAAACGCTTAGCCTAGCCACACCCCC[illegible]CGGGAAACAGCAGTGATTAACCTT[illegible]AGCAATAA
ACGAAAGTTTAACTAAGCTATACTAAC[illegible]CCAGGGTTGGTCAATTTCGTGC[illegible]AGCCACCGC
GGTCACACGATTAACCCAAGTCAATAGA[illegible]GCCGGCGTAAAGAGTGTTTT[illegible]GATCACCCCC
TCCCCAATAAAGCTAAAACTCACCTGAGTT[illegible]TAAAAAACTCCAGTT[illegible]AATAGAC
TACGAAAGTGGCTTTAACATATCTGAACACACAA[illegible]CCCAA[illegible]TTAGA
TACCCCACTATGCTTAGCCCTAAACCTCAACAGTTAAATCAACAAAACT[illegible]GAA
CACTACGAGCCACAGCTTAAAACTCAAAGGACCTGGCGGTGCTTCATATCC[illegible]G
AGCCTGTTCTGTAATCGATAAACCCCGATCAACCTCACCACCTCTTGCTCAG[illegible]
CCGCCATCTTCAGCAAACCCTGATGAAGGCTACAAAGTAAGCGCAAGTACCCAC[illegible]
ACGTTAGGTCAAGGTGTAGCCCATGAGGTGGCAAGAAATGGGCTACATTTTCTACC[illegible]
AAAACTACGATAGCCCTTATGAAACTTAAGGGTCGAAGGTGGATTTAGCAGTAAACTA[illegible]
AGTAGAGTGCTTAGTTGAACAGGGCCCTGAAGCGCGTACACACCGCCCGTCACCCTCCTC
AAGTATACTTCAAAGGACATTTAACTAAAACCCCTACGCATTTATATAGAGGAGACAAGT
CGTAACCTCAAACTCCTGCCTTTGGTGATCCACCCGCCTTGGCCTACCTGCATAATGAAG
AAGCACCCAACTTACACTTAGGAGATTTCAACTTAACTTGACCGCTCTGAGCTAAACCTA
GCCCCAAACCCACTCCACCTTACTACCAGACAACCTTAGCCAAACCATTTACCCAAATAA
AGTATAGGCGATAGAAATTGAAACCTGGCGCAATAGATATAGTACCGCAAGGGAAAGATG
AAAAATTATAACCAAGCATAATATAGCAAGGACTAACCCCTATACCTTCTGCATAATGAA
TTAACTAGAAATAACTTTGCAAGGAGAGCCAAAGCTAAGACCCCCGAAACCAGACGAGCT
ACCTAAGAACAGCTAAAAGAGCACACCCGTCTATGTAGCAAAATAGTGGGAAGATTTATA
GGTAGAGGCGACAAACCTACCGAGCCTGGTGATAGCTGGTTGTCCAAGATAGAATCTTAG
TTCAACTTTAAATTTGCCCACAGAACCCTCTAAATCCCCTTGTAAATTTAACTGTTAGTC
CAAAGAGGAACAGCTCTTTGGACACTAGGAAAAAACCTTGTAGAGAGAGTAAAAAATTTA
```

Likelihood of random seed string:

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

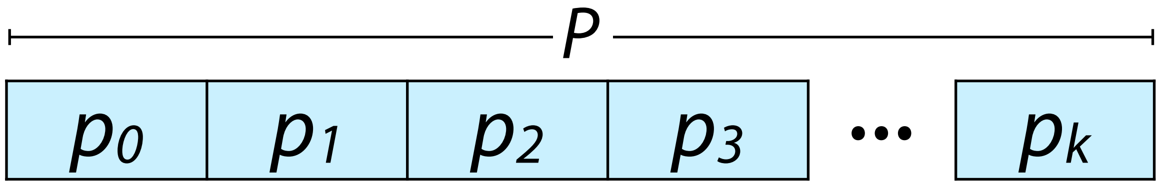
\# of times seed will occur by chance in T:

Likelihood * (~ length)

$$2.48e - 15$$

Approximate Pattern Matching

"Seed and extend" approach to pattern matching



P

p_0 p_1 p_2 p_3 … p_k

Advantages

Reuse favorite exact matching algs; 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

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.

$P =$ A A A A A A A A $\quad d = 3$

How many partitions?

What is the characters in each partition(s)?

$T =$ ... B B B B B B A A B B B B B B ...

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.

$P =$ B A A A A A A A $\quad d = 1$

$T =$ ... B B B B B B A A A A A B A A ...

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?

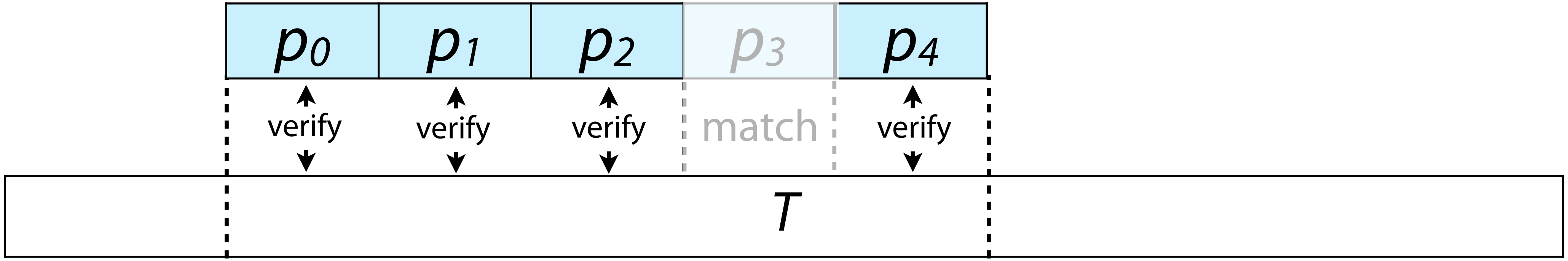
`vector<int> approximate_search(fname, P, mm)`

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



`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} }`

`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: **aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa** $k = 4$

aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa

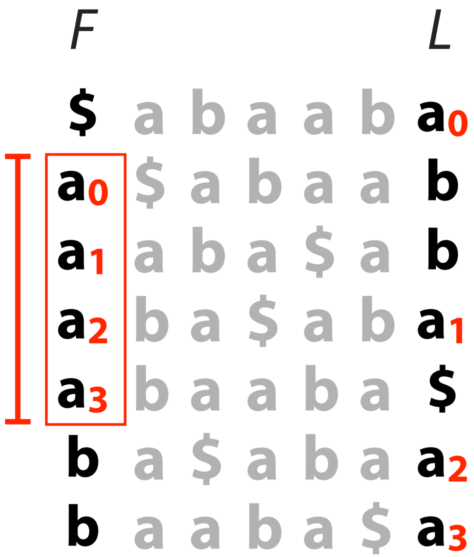
Bonus Slides

FM Index w/ mismatches

Start with shortest suffix, then match successively longer suffixes

Keep track of mismatches for each suffix

P = **aba**

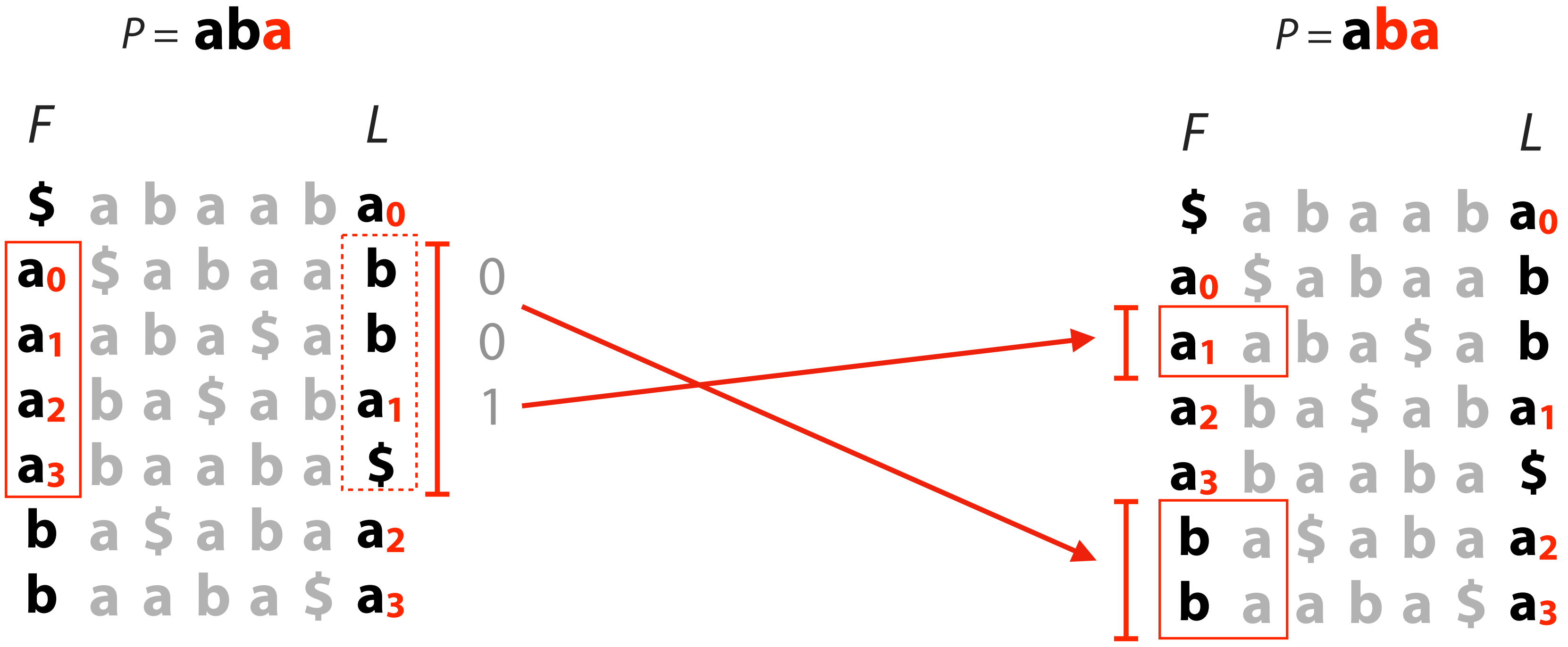


F						L
$	a	b	a	a	b	a_0
a_0	$	a	b	a	a	b
a_1	a	b	a	$	a	b
a_2	b	a	$	a	b	a_1
a_3	b	a	a	b	a	$
b	a	$	a	b	a	a_2
b	a	a	b	a	$	a_3

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

FM Index w/ mismatches

We have rows beginning with **a**, now we want rows beginning with **ba**



P = **aba**

F						*L*	
$	a	b	a	a	b	a_0	
a_0	$	a	b	a	a	b	0
a_1	a	b	a	$	a	b	0
a_2	b	a	$	a	b	a_1	1
a_3	b	a	a	b	a	$	
b	a	$	a	b	a	a_2	
b	a	a	b	a	$	a_3	

P = **aba**

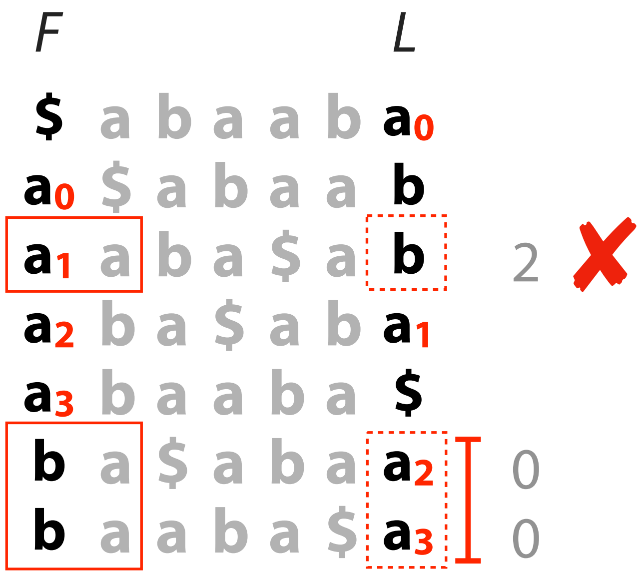
F						*L*
$	a	b	a	a	b	a_0
a_0	$	a	b	a	a	b
a_1	a	b	a	$	a	b
a_2	b	a	$	a	b	a_1
a_3	b	a	a	b	a	$
b	a	$	a	b	a	a_2
b	a	a	b	a	$	a_3

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 = **aba**



$F \qquad\qquad L$

$\$\ a\ b\ a\ a\ b\ a_0$

$a_0\ \$\ a\ b\ a\ a\ b$

$a_1\ a\ b\ a\ \$\ a\ b \quad 2$ ✗

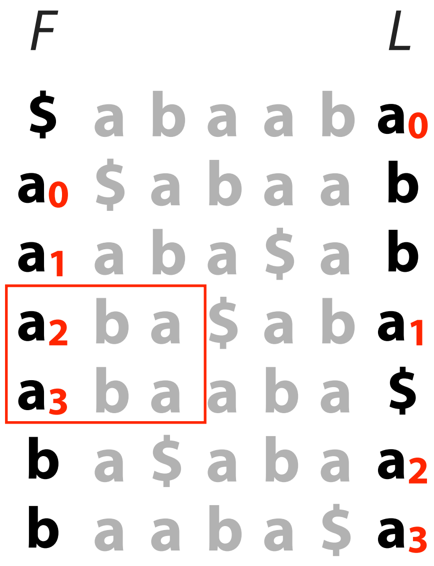
$a_2\ b\ a\ \$\ a\ b\ a_1$

$a_3\ b\ a\ a\ b\ a\ \$$

$b\ a\ \$\ a\ b\ a\ a_2 \quad 0$

$b\ a\ a\ b\ a\ \$\ a_3 \quad 0$

P = **aba**



$F \qquad\qquad L$

$\$\ a\ b\ a\ a\ b\ a_0$

$a_0\ \$\ a\ b\ a\ a\ b$

$a_1\ a\ b\ a\ \$\ a\ b$

$a_2\ b\ a\ \$\ a\ b\ a_1$

$a_3\ b\ a\ a\ b\ a\ \$$

$b\ a\ \$\ a\ b\ a\ a_2$

$b\ a\ a\ b\ a\ \$\ a_3$

Only works for Hamming Distance (mismatches)!