Exact pattern matching & string search

UNIVERSITY OF
MARYLAND

Why Exact Matching?

As *loose* motivation, consider the problem of mapping a read r to the genome G.

In reality, we would not use exact matching for this; why?

However, exact matching is useful here:

- Find all places where a substring of the query matches the reference exactly (seeds)
- Filter out regions with insufficient exact matches to warrant further investigation
- Perform a “constrained” alignment that includes these exact matching “seeds”

Requires efficient exact search

Here is where we use efficient algorithms for inexact matching (alignment)

Exact String Matching Problem

Today, we'll talk about exact matching algorithms that are **quadratic** (no better than alignment!) and **linear**. Then we'll start talking about ***much*** faster approaches, but they require pre-processing the reference.

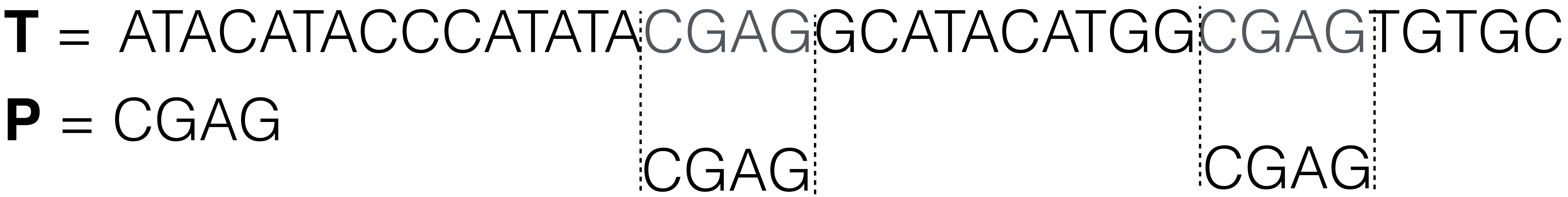
Exact String Matching Problem

Given: A string **T** (called the *text*) and a string **P** (called the *pattern*).

Find: All occurrences of **P** in **T**.

|**T**| > |**P**|

An *occurrence* of **P** in **T** is a substring of **T** equal to **P**



T = ATACATACCCATATACGAGGCATACATGGCGAGTGTGC

P = CGAG

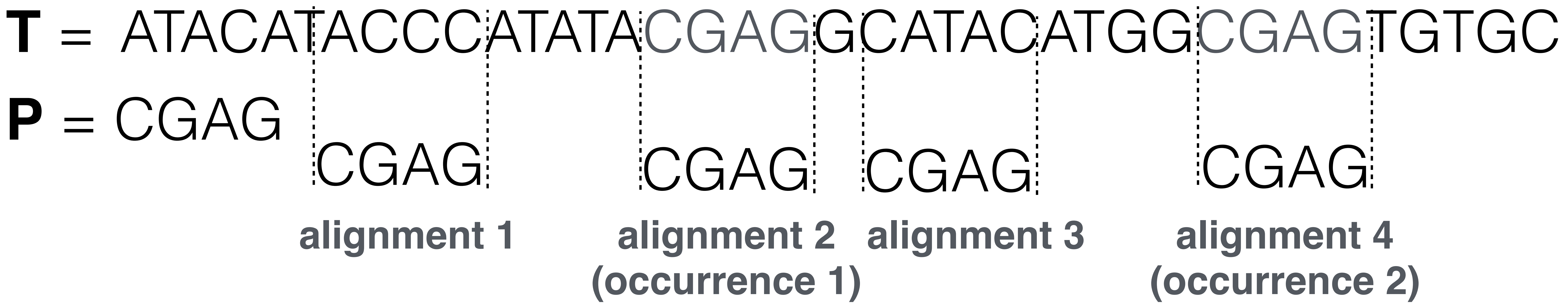
CGAG

CGAG

Occurrences vs. Alignments

An *alignment* of **P** to **T** is a correspondence (not necessarily an occurrence) between a substring of **T** and **P**

all occurrences are alignments but not all alignments are occurrences



T = ATACATACCCATATACGAGGCATACATGGCGAGTGTGC

P = CGAG

Occurrences vs. Alignments

How many possible *alignments* of **P** are there in **T**?

```
T = ATACATACCCATATACGAGGCATACATGGCGAGTGTGC
P = CGAG
     CGAG
      CGAG
       CGAG
        CGAG
           ⋱
```

Occurrences vs. Alignments

How many possible *alignments* of **P** are there in **T**?

T = ATACATACCCATATACGAGGCATACATGGCGAGTGTGC

P = CGAG

CGAG

CGAG

CGAG

CGAG

⋱

$|\mathbf{T}| - |\mathbf{P}| + 1$

A naive algorithm

What is the simplest algorithm you can think of to solve the exact string matching problem?

Seriously, I’m not going to change the slide until somebody suggests something really naive!

A naive algorithm

Naive algorithm 1: Consider all alignments of **P** to **T**, and report each alignment that is an occurrence.

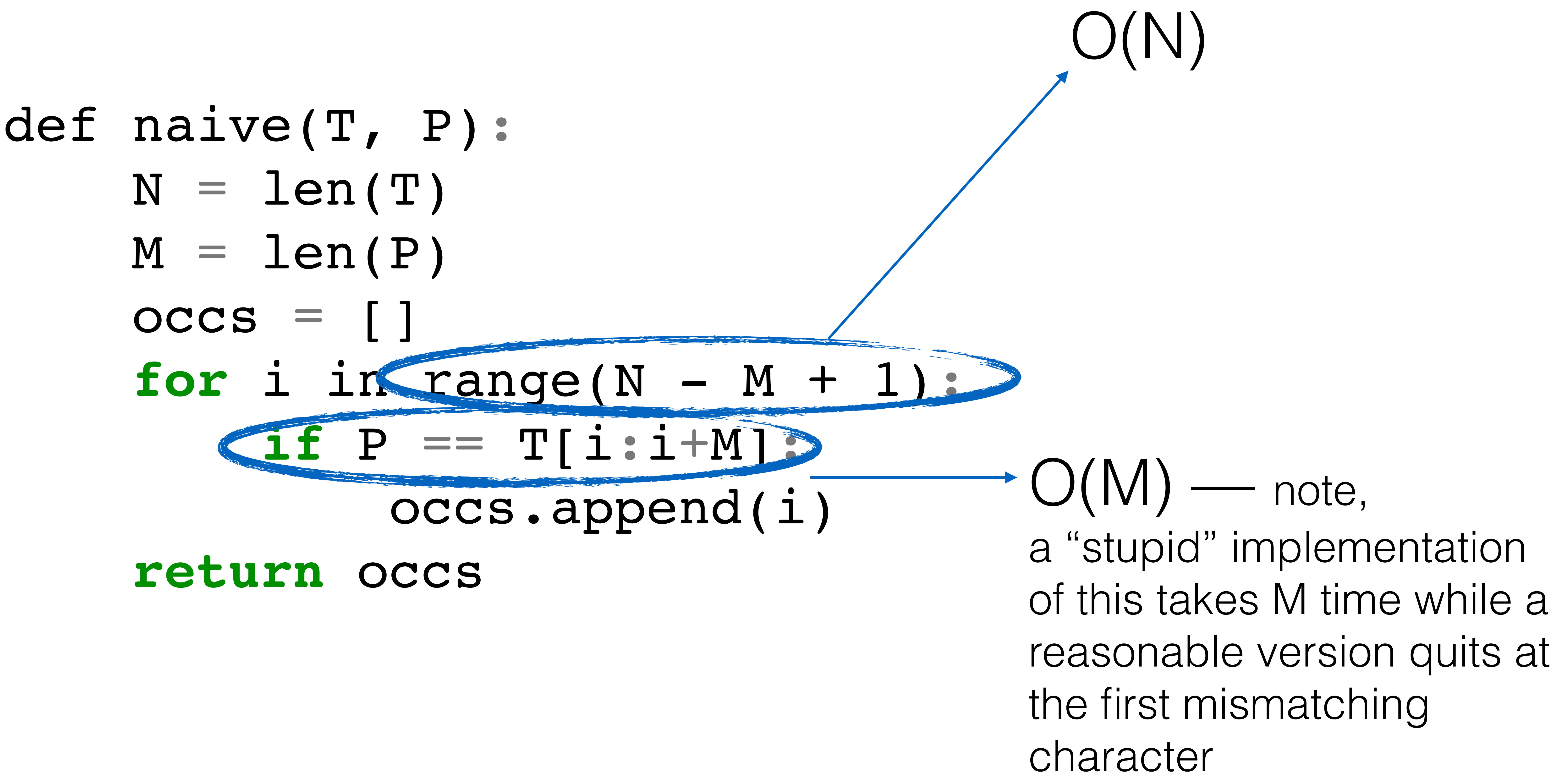
```
def naive(T, P):
    N = len(T)
    M = len(P)
    occs = []
    for i in range(N - M + 1):
        if P == T[i:i+M]:
            occs.append(i)
    return occs
```

A naive algorithm

```
def naive(T, P):
    N = len(T)
    M = len(P)
    occs = []
    for i in xrange(N - M + 1):
        if P == T[i:i+M]:
            occs.append(i)
    return occs
```

Worst-case Runtime?

A naive algorithm



```
def naive(T, P):
    N = len(T)
    M = len(P)
    occs = []
    for i in range(N - M + 1):
        if P == T[i:i+M]:
            occs.append(i)
    return occs
```

O(N)

O(M) — note, a “stupid” implementation of this takes M time while a reasonable version quits at the first mismatching character

O(N) * O(M) = O(NM) time

A naive algorithm

Best scenario for naive:

T: GAGAGGAGTTATATATGAATAGAGATAGAGACGAG

P: CGAG

Because every alignment but the last disagrees
on the very first character, the inner loop takes O(1) time,
except for the single match which takes O(M) time
O(N+M)

A naive algorithm

Worst scenario for naive:

T: CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

P: CCCCG

Because every alignment is a match for
P, the inner loop requires M char. compares each time
O(NM)

A naive algorithm

There's a **big** gap between

The best case time for naive O(N+M) and
The worst case time for naive O(NM)

How can we improve the worst case time?

Can we devise a method that is O(N+M) even in the worst case?

Z boxes and the Z algorithm

T: ATACGGCACATACCATACGAATATACAAA

Def: Let Z_i be the length of the longest substring *starting at* i that matches a prefix of T.

Z boxes and the Z algorithm

```
 T:   ATACGGCACATACCATACGAATATATACAAA
Z_T: -101000001040100501001304010111
```

Def: Let Z_i be the length of the longest substring *starting at* i that matches a prefix of T.

Z boxes and the Z algorithm

T: ATACGGCACATACCATACGAATATACAAA

Z_T: -10100001040100501001304010111

Def: Let Z_i be the length of the longest substring *starting at* i that matches a prefix of T.

Naïvely, there is an O(?) algorithm to compute the z values

Z boxes and the Z algorithm

```
 T:   ATACGGCACATACCATACGAATATACAAA
Z_T: -10100001040100501001304010111
```

Def: Let Z_i be the length of the longest substring *starting at* i that matches a prefix of T.

Naïvely, there is an $O(T^2)$ algorithm to compute the z values

Ignore this complexity for a second; **how could we use z values to solve exact pattern matching**?

Z boxes and the Z algorithm

P: ACA

T: ATACGGCACATACCATACGAATATACAAA

Def: Let Z_i be the length of the longest substring *starting at* i in T that matches a prefix of P.

Ignore this complexity for a second; **how could we use z values to solve exact pattern matching**?

Z boxes and the Z algorithm

P$T: ACA$ATACGGCACATACCATACGAATATACAAA

Def: Let Z_i be the length of the longest substring *starting at* i in T that matches a **prefix of P**.

Now, any Z_i value = |P| designates that an occurrence of P exists at position i in T.

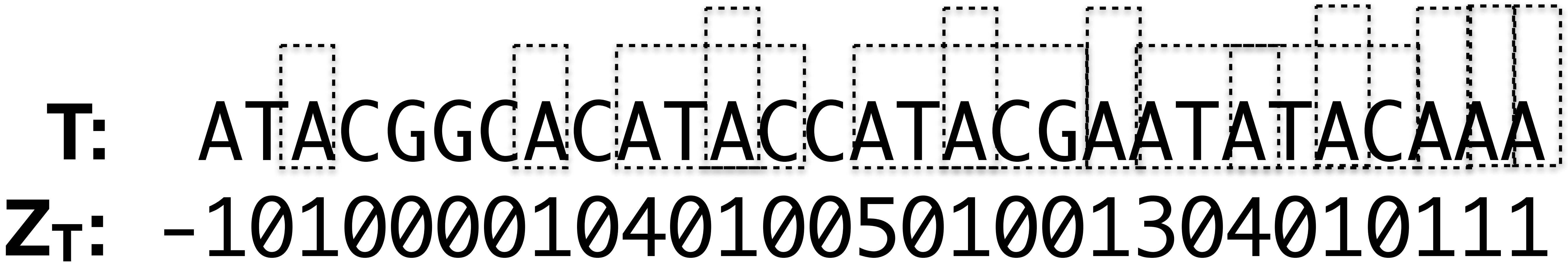
Note: $\$ \notin \Sigma$ ensures that Z_i is always ≤ |P|

Z boxes and the Z algorithm

P$T: ACA$ ATACGGCACATACCATACGAATATACAAA

Now that the longest possible Z_i is ≤ |P| then we are back to an O(|T| |P|) algorithm … back to the problem at hand; how do we make this better?

Z boxes (boxen?)

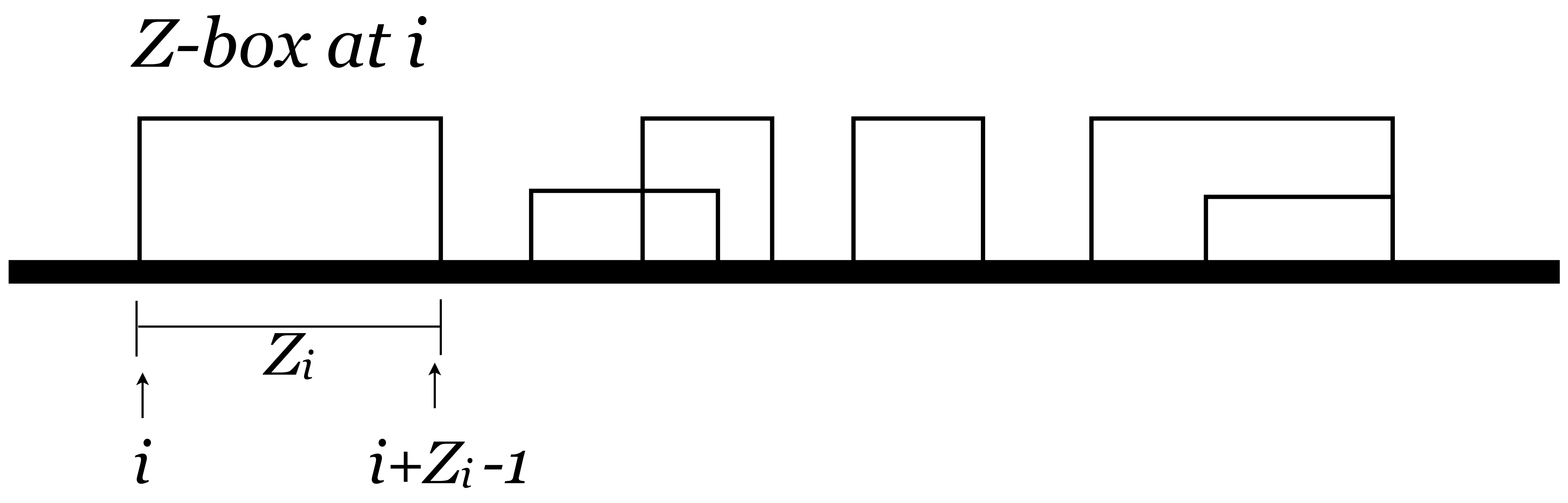


T: ATACGGCACATACCATACGAATATACAAA

Z_T: -10100001040100501001304010111

Imagine a “box” (possibly of length 0) starting at every position. The left-most end of the box is where the match with the prefix begins, and each box extends Z_i characters to the right (to position $i + Z_i - 1$).

Z Boxes



Def. *Z-box at i* is the substring starting at i and continuing to $i+Z_i-1$. This is the substring that matches the prefix. There is no Z-box at i if $Z_i = 0$.

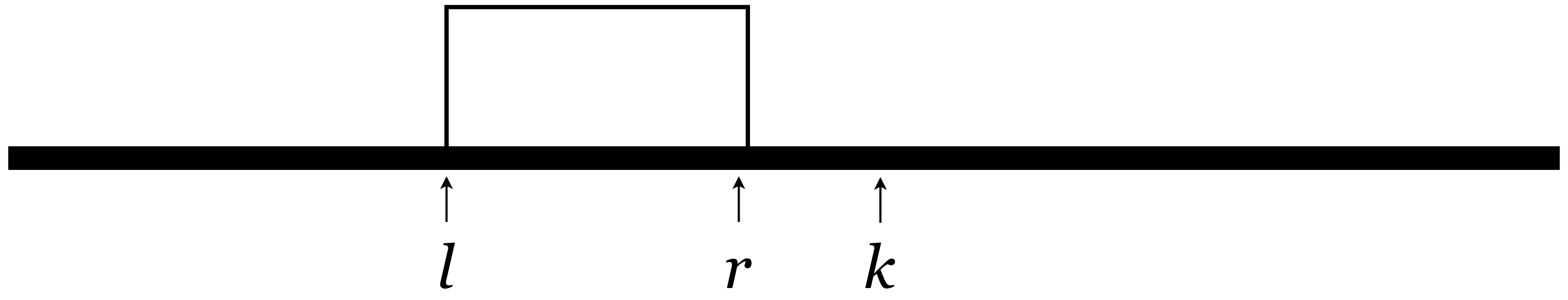
- Algorithm for computing Z_i will iteratively compute Z_k given:
  - $Z_2 \ldots Z_{k-1}$, and
  - the boundaries l, r of the rightmost Z-box found starting someplace in $2 \ldots k-1$.
  - you don't need l to understand how the algorithm works, but it is required to efficiently compute the necessary quantities

Slide adapted from material by Carl Kingsford

Z Algorithm

- Input: $Z_2 \ldots Z_{k-1}$, and the boundaries *l, r* of the rightmost Z-box found starting someplace in 2...*k*-1.
- Output: Z_k, and updated *l, r*

1. If $k > r$, explicitly compute Z_k by comparing with prefix.
   If $Z_k > 0$: $l = k$ and $r = k + Z_k - 1$ (since this is a new farther right Z-box).



The current index is *beyond* the bound of the rightmost z-box.

The structure of the rightmost z-box can not tell us what to expect for Z_k

Compute Z_k by explicit comparison and update *l*,*r* if $Z_k > 0$

Slide adapted from material by Carl Kingsford

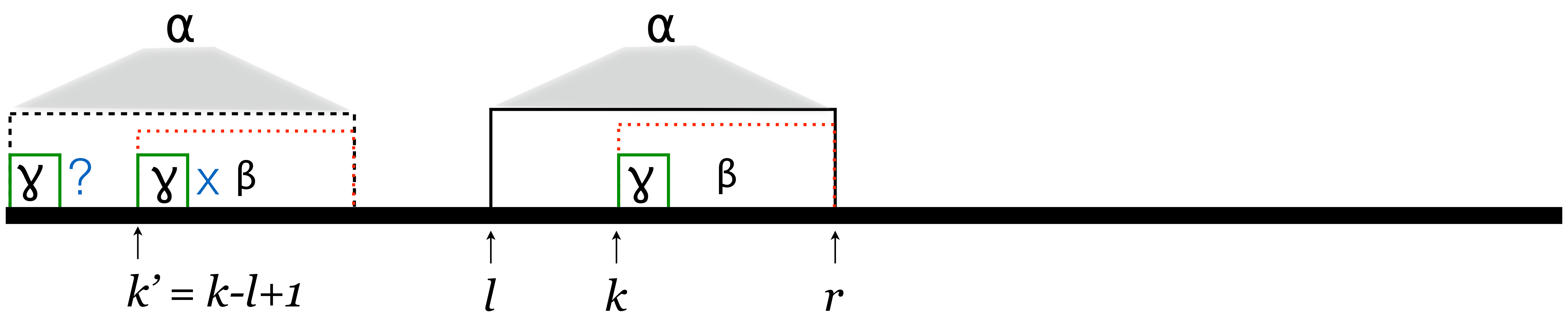
Z Algorithm

- Input: $Z_2 \ldots Z_{k-1}$, and the boundaries *l, r* of the rightmost Z-box found starting someplace in 2…*k*-1.
- Output: Z_k, and updated *l, r*

If $k \leq r$, this is the situation:

Case 2a : $Z_{k'} < |\beta|$:

$Z_{k'} < |\beta|$: Then the ɣ that is a prefix of β is also a prefix of α, **but** the character occurring after the ɣ starting at k' is *not* the same as the character after the ɣ starting at the beginning of the string … why?



Slide adapted from material by Carl Kingsford

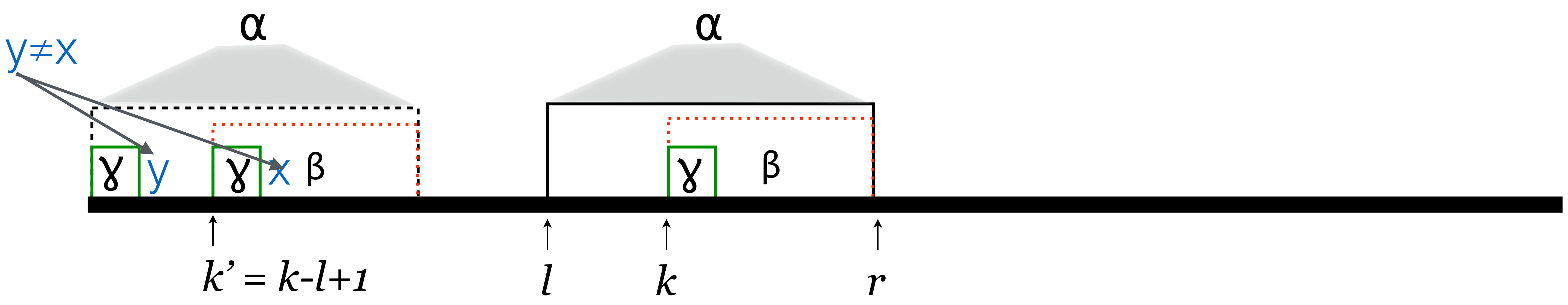
Z Algorithm

- Input: $Z_2 \ldots Z_{k-1}$, and the boundaries *l, r* of the rightmost Z-box found starting someplace in 2…*k*-1.
- Output: Z_k, and updated *l, r*

If $k \leq r$, this is the situation:

Case 2a : $Z_{k'} < |\beta|$:

$Z_{k'} < |\beta|$: Then the ɣ that is a prefix of β is also a prefix of α, **but** the character occurring after the ɣ starting at k' is *not* the same as the character after the ɣ starting at the beginning of the string … **why**?



Slide adapted from material by Carl Kingsford

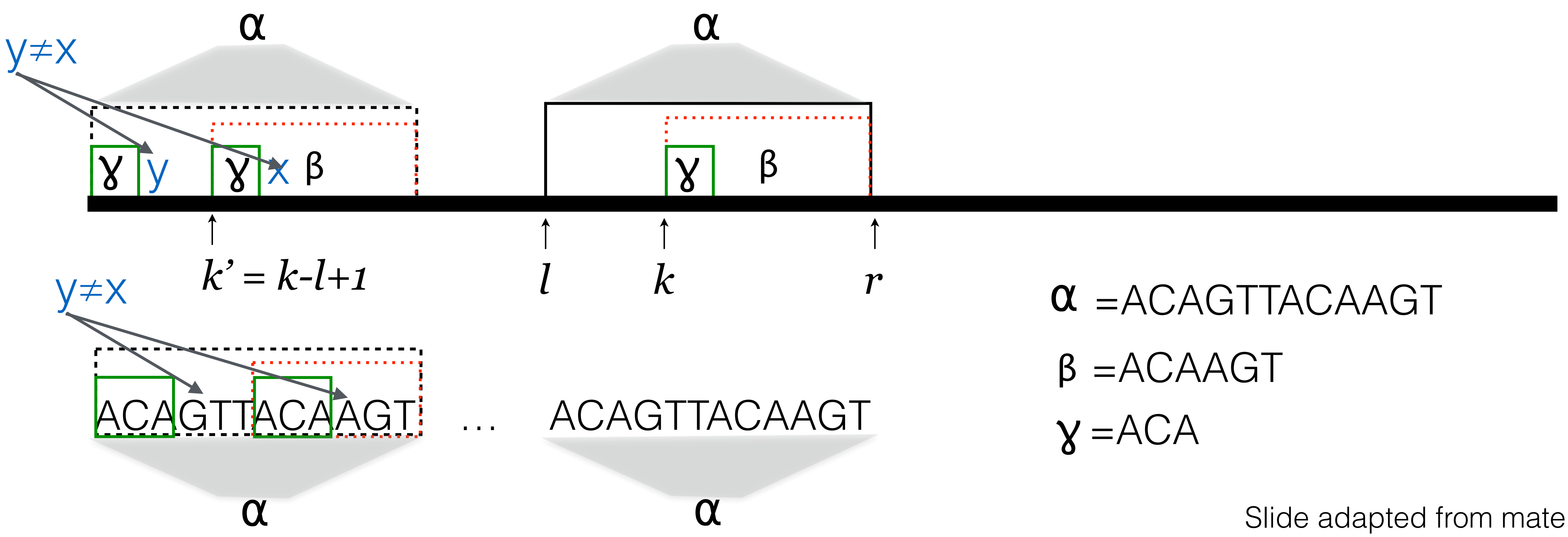
Z Algorithm

- Input: $Z_2 \ldots Z_{k-1}$, and the boundaries *l, r* of the rightmost Z-box found starting someplace in 2…*k*-1.
- Output: Z_k, and updated *l, r*

If $k \le r$, this is the situation:

Case 2a : $Z_{k'} < |\beta|$:

$Z_{k'} < |\beta|$: Then the ɣ that is a prefix of β is also a prefix of α, **but** the character occurring after the ɣ starting at k' is *not* the same as the character after the ɣ starting at the beginning of the string … **why**?



e.g.

α =ACAGTTACAAGT

β =ACAAGT

ɣ =ACA

Slide adapted from material by Carl Kingsford

Z Algorithm

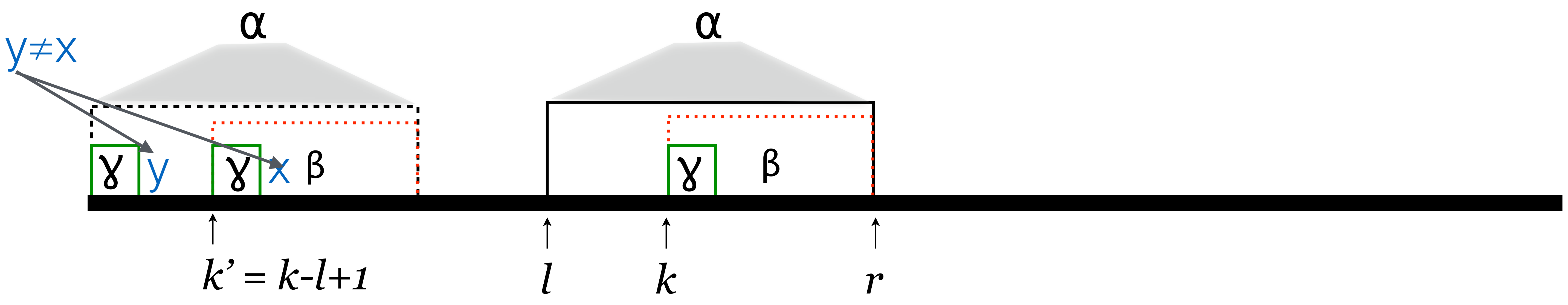
- Input: $Z_2 \ldots Z_{k-1}$, and the boundaries *l, r* of the rightmost Z-box found starting someplace in 2…*k*-1.
- Output: Z_k, and updated *l, r*

If $k \leq r$, this is the situation:

Case 2a : $Z_{k'} < |\beta|$:

$Z_{k'} < |\beta|$: Then the ɣ that is a prefix of β is also a prefix of α, **but** the character occurring after the ɣ starting at k’ is *not* the same as the character after the ɣ starting at the beginning of the string … **why**?

If x = y, then $Z_{k'} > |ɣ|$, because the shared prefix starting at k’ and 0 would had to have been longer.



Slide adapted from material by Carl Kingsford

Z Algorithm

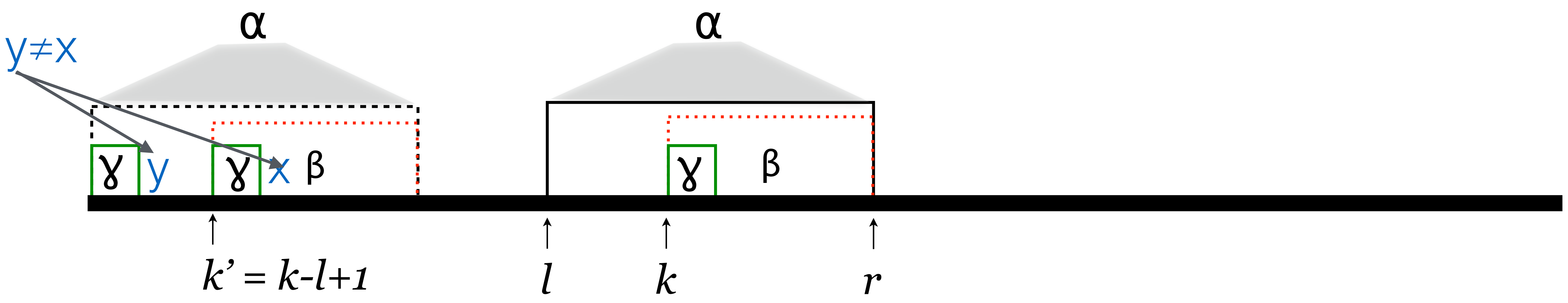
- Input: $Z_2 \ldots Z_{k-1}$, and the boundaries *l, r* of the rightmost Z-box found starting someplace in 2…*k*-1.
- Output: Z_k, and updated *l, r*

If $k \leq r$, this is the situation:

Case 2a : $Z_{k'} < |\beta|$:

$Z_{k'} < |\beta|$: Then the ɣ that is a prefix of β is also a prefix of α, **but** the character occurring after the ɣ starting at k' is *not* the same as the character after the ɣ starting at the beginning of the string … **why**?

If x = y, then $Z_{k'} > |\gamma|$, because the shared prefix starting at k' and 0 would had to have been longer. But β = β, so $Z_k = Z_{k'}$



In this case, set $Z_k = Z_{k'}$ and leave *l, r* unchanged.

Slide adapted from material by Carl Kingsford

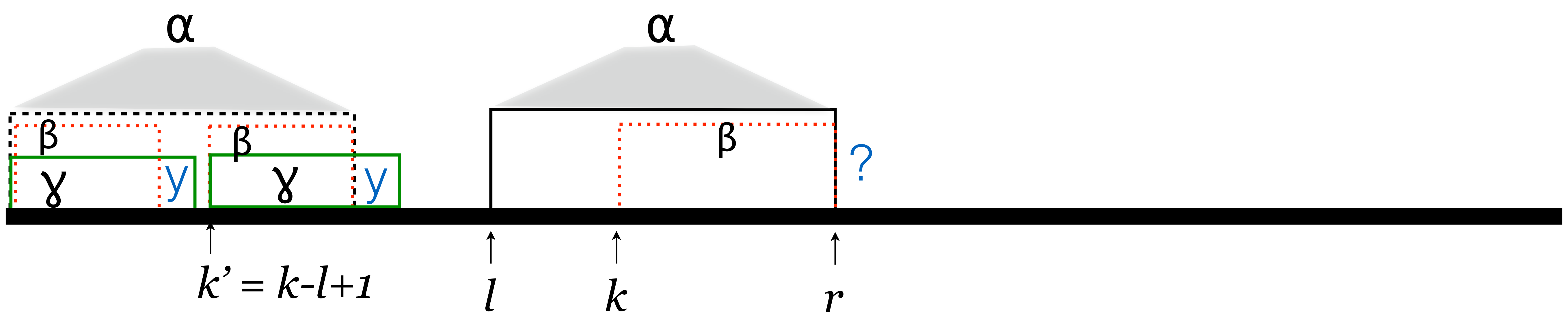
Z Algorithm

- Input: $Z_2 \ldots Z_{k-1}$, and the boundaries *l, r* of the rightmost Z-box found starting someplace in 2…*k*-1.
- Output: Z_k, and updated *l, r*

If $k \leq r$, this is the situation:

Case 2b : $Z_{k'} > |\beta|$:

$Z_{k'} > |\beta|$: Then the ɣ that starts at k' matches the ɣ that starts at the beginning of T, **but**, it cannot (completely) match the substring starting at k … why?



Slide adapted from material by Carl Kingsford

Z Algorithm

- Input: $Z_2 \ldots Z_{k-1}$, and the boundaries *l, r* of the rightmost Z-box found starting someplace in 2…*k*-1.
- Output: Z_k, and updated *l, r*

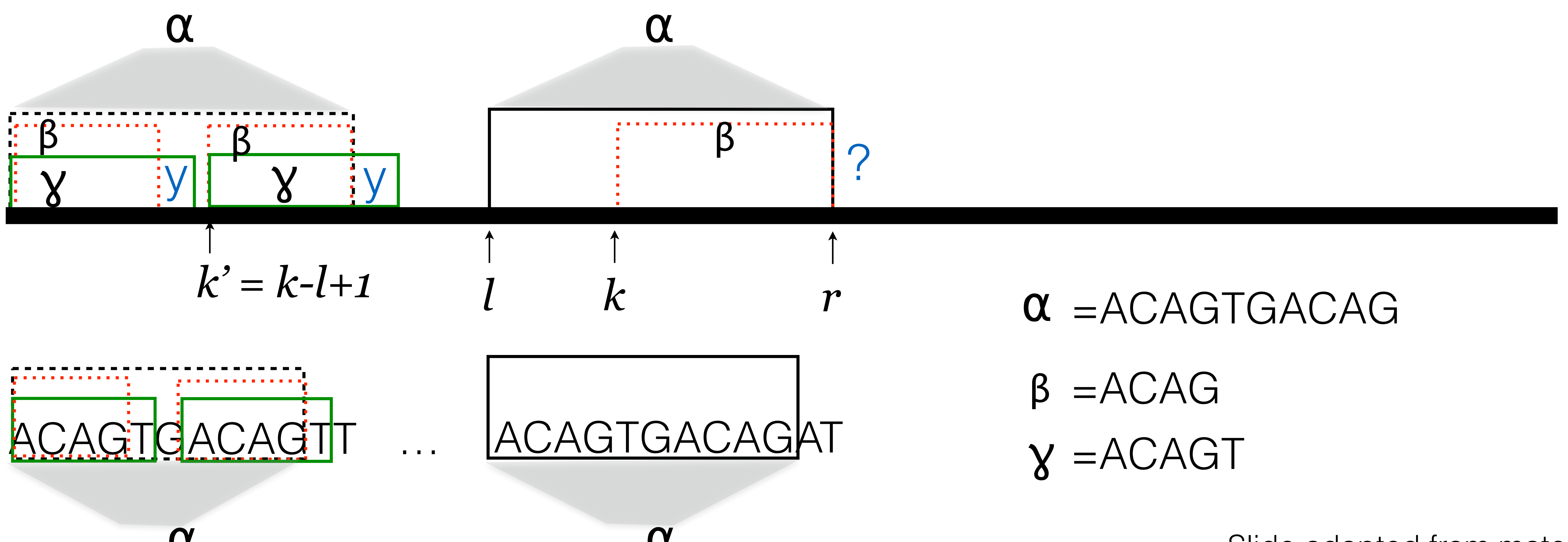
If $k \leq r$, this is the situation:

Case 2b : $Z_{k'} > |\beta|$:

$Z_{k'} > |\beta|$: Then the ɣ that starts at k' matches the ɣ that starts at the beginning of T, **but**, it cannot (completely) match the substring starting at k … why?

Note: Here, we are not necessarily saying that the "Z-box" starting at 0 (ill-defined anyway) is of length |α|; Rather α is defined by Z_l

e.g.



α =ACAGTGACAG

β =ACAG

ɣ =ACAGT

Slide adapted from material by Carl Kingsford

Z Algorithm

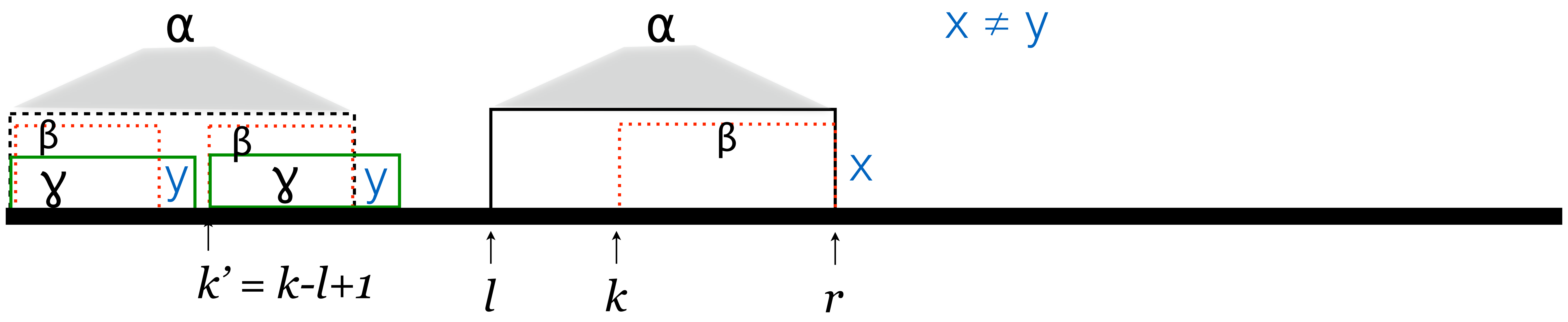
- Input: $Z_2 \ldots Z_{k-1}$, and the boundaries *l, r* of the rightmost Z-box found starting someplace in 2…*k*-1.
- Output: Z_k, and updated *l, r*

If $k \leq r$, this is the situation:

Case 2b : $Z_{k'} > |\beta|$:

$Z_{k'} > |\beta|$: Then the ɣ that starts at k' matches the ɣ that starts at the beginning of T, **but**, it cannot (completely) match the substring starting at k … why?

If it did, then the z-box starting at position *l*, would be longer (extend past *r*), contradicting the fact that Z_l is the *longest* substring starting at l that matches a prefix of T.



Set $Z_k = |\beta|$ and leave *l, r* unchanged.

Slide adapted from material by Carl Kingsford

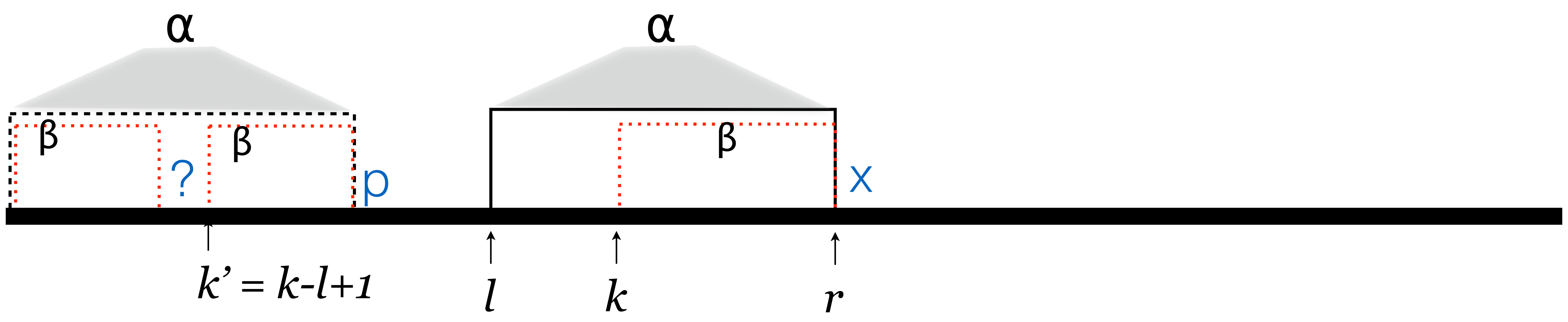
Z Algorithm

- Input: $Z_2 \ldots Z_{k-1}$, and the boundaries *l, r* of the rightmost Z-box found starting someplace in 2…*k*-1.
- Output: Z_k, and updated *l, r*

If $k \leq r$, this is the situation:

Case 2c : $Z_{k'} = |\beta|$:

$Z_{k'} = |\beta|$: Then the character following the z-box of $Z_{k'}$, cannot be the same as the character following the length β prefix of the string … why?



Slide adapted from material by Carl Kingsford

Z Algorithm

- Input: $Z_2 \ldots Z_{k-1}$, and the boundaries *l, r* of the rightmost Z-box found starting someplace in 2…*k*-1.
- Output: Z_k, and updated *l, r*

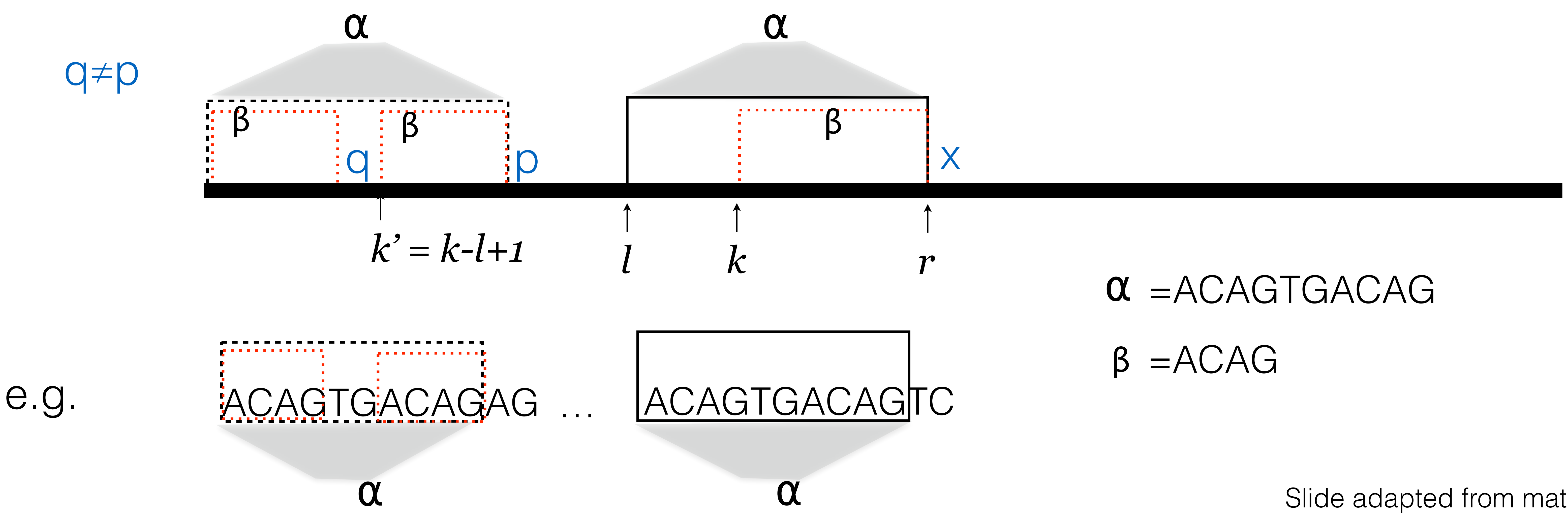
If $k \leq r$, this is the situation:

Case 2c : $Z_{k'} = |\beta|$:

$Z_{k'} = |\beta|$: Then the character following the z-box of $Z_{k'}$, cannot be the same as the character following the length β prefix of the string … why?

If q = p, then $Z_{k'}$ would have length > |β|

What do we know about x … x ≠ p. Is x = q?



α =ACAGTGACAG

β =ACAG

Slide adapted from material by Carl Kingsford

Z Algorithm

- Input: $Z_2 \ldots Z_{k-1}$, and the boundaries *l, r* of the rightmost Z-box found starting someplace in 2…*k*-1.
- Output: Z_k, and updated *l, r*

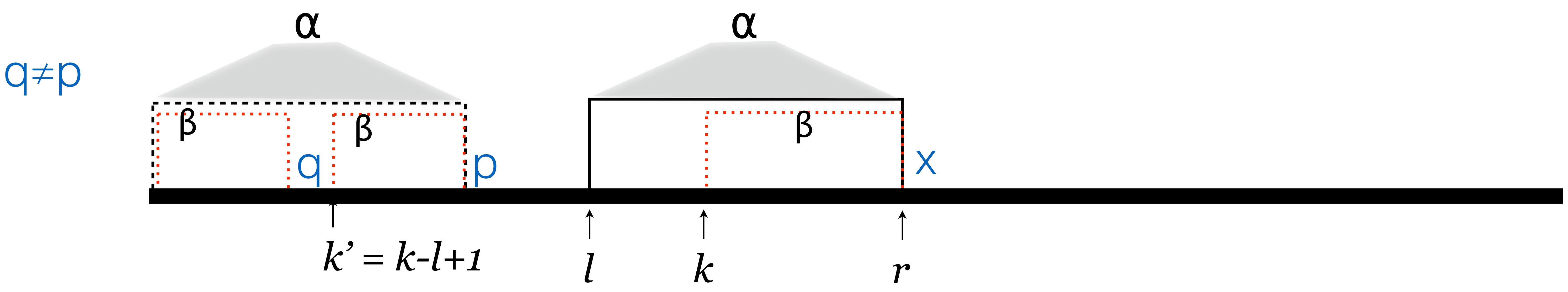
If $k \leq r$, this is the situation:

Case 2c : $Z_{k'} = |\beta|$:

$Z_{k'} = |\beta|$: Then the character following the z-box of $Z_{k'}$, cannot be the same as the character following the length β prefix of the string … why?

If q = p, then $Z_{k'}$ would have length > |β|

What do we know about x … x ≠ p. Is x = q? We don't know! Must check.



Explicitly compare <u>after *r*</u> to set Z_k. $l = k$, r = point where comparison failed

Slide adapted from material by Carl Kingsford

Analysis

- Correctness follows by induction and the arguments we made in the description of the algorithm.
    - If you follow all of the sub-cases, the correctness of z-alg is implied
- Runs in O(|P|+|T|) time:
    - only match characters covered by a Z-box once, so there are O(|P|+|T|) matches.
    - every iteration contains at most one mismatch, so there are O(|P|+|T|) mismatches.
- Immediately gives an O(|P| + |T|)-time algorithm for string matching as described a few slides ago.
    - O(|P| + |T|) is the best possible worst-case running time, since you might have to look at the whole input.
    - But better algorithms exist in practice that, for real instances, have expected sublinear runtime.

Slide adapted from material by Carl Kingsford

Summary

The pattern matching problem seeks to find all occurrences of a pattern P in a text T

The naive algorithm for the problem takes O(MN) time

By exploiting structure in the *pattern*, we reduce the worst case runtime to O(M+N)