

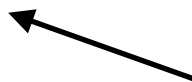

Exact pattern matching & string search

Why Exact Matching?

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

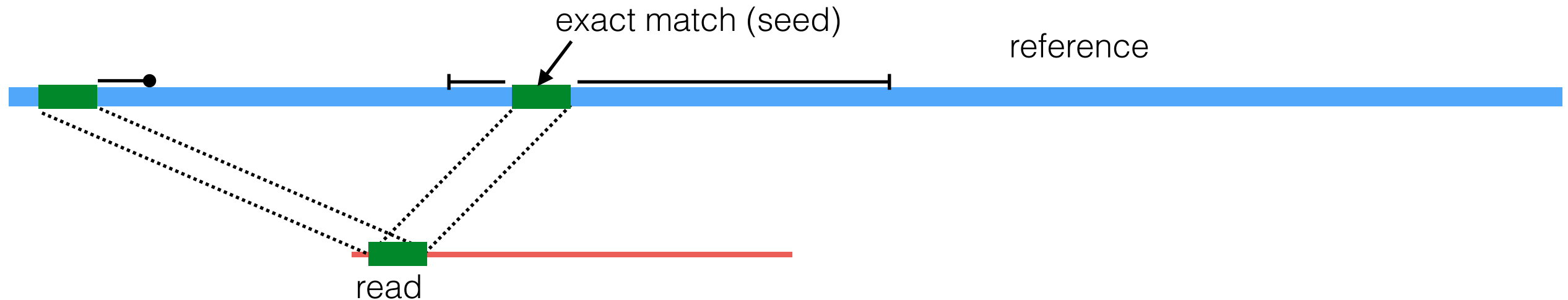
In reality, we would not use exact matching for this; why?

However, exact matching is useful here:

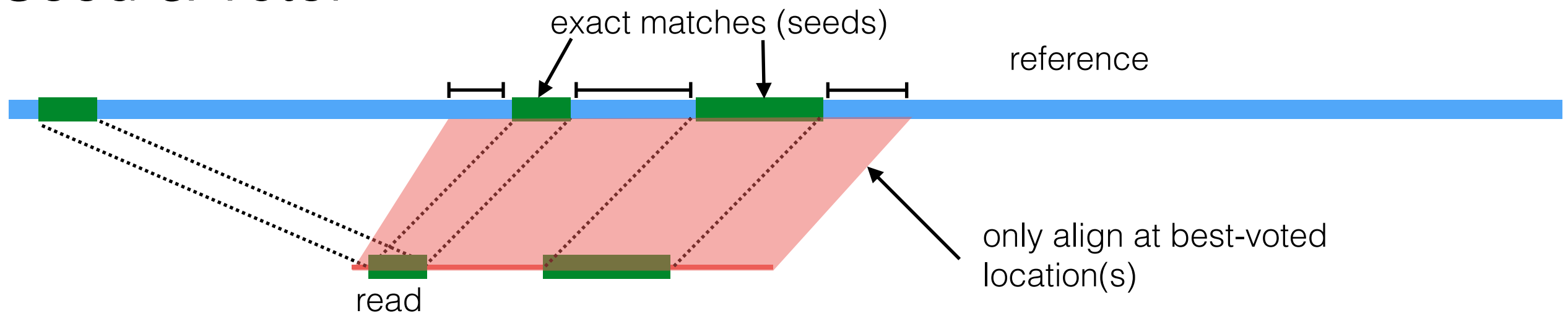
- Find all places where a substring of the query matches the reference exactly (seeds)  Requires efficient exact search
- Filter out regions with insufficient exact matches to warrant further investigation
- Perform a “constrained” alignment that includes these exact matching “seeds”  Here is where we use our alignment DPs

Typical Strategies

Seed & Extend:



Seed & Vote:



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**.

$$|\mathbf{T}| > |\mathbf{P}|$$

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

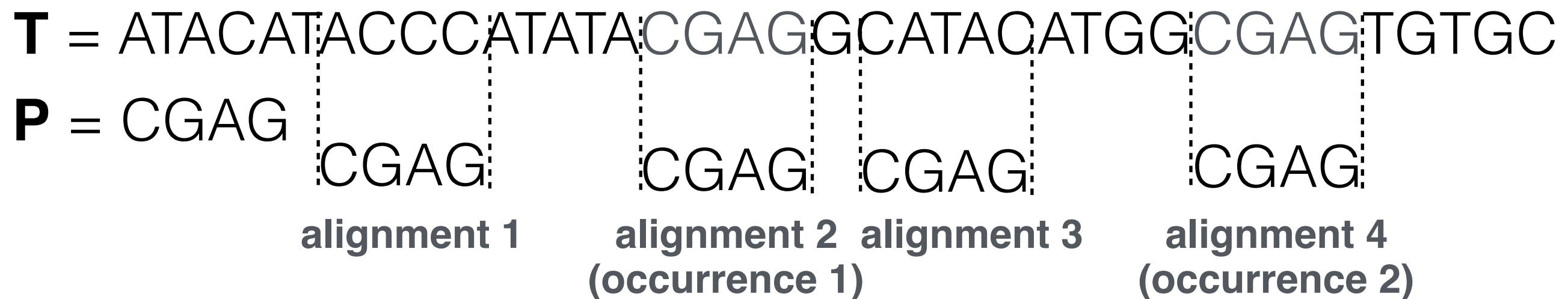
T = ATACATACCCATATACGAGGCATACATGGCGAGTGTGC
P = CGAG



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



Occurrences vs. Alignments

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

T = ATACATACCCATATACGAGGCGATACATGGCGAGTGTGC

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: CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

P: C C C C G

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: ATACGGGCACATACCATACGAATATACAAA

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: ATACGGGCACATACCATACGAATATACAAA

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: ATACGGGCACATACCATACGAATATACAAA

Z_T: -1 0 1 0 0 0 0 1 0 4 0 1 0 0 5 0 1 0 0 1 3 0 4 0 1 0 1 1 1

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: ATACGGGCACATACCATACGAATATACAAA
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 $O(n^2)$ algorithm to compute the z values

Z boxes and the Z algorithm

T: ATACGGGCACATACCATACGAATATACAAA
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 $\leq |P|$

Z boxes and the Z algorithm

P\$T: ACA\$ATACGGCACATACCATACGAATATACAAA

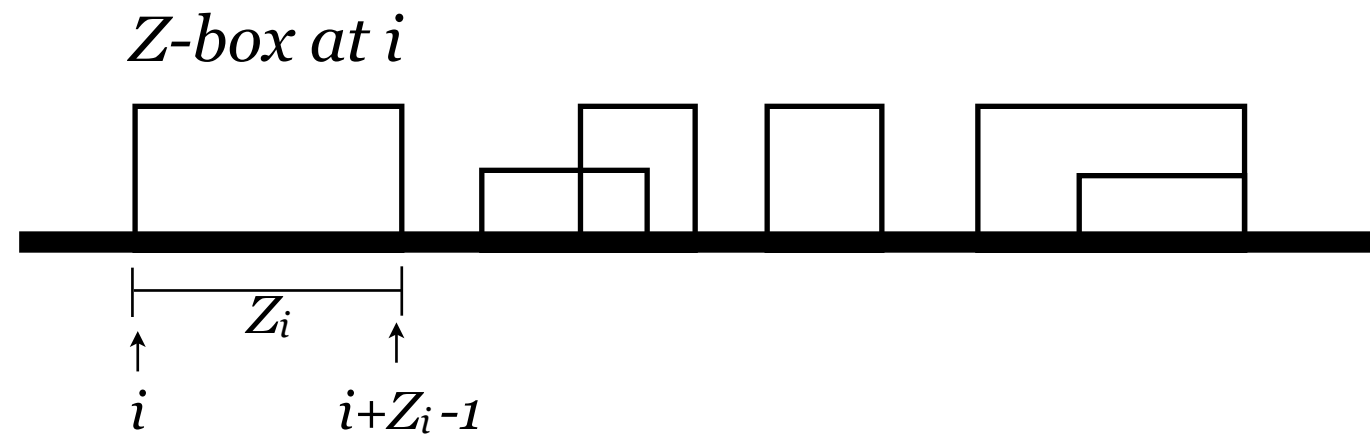
Now that the longest possible Z_i is $\leq |P|$ then we are back to an $O(|T| |P|)$ algorithm ... back to the problem at hand; how do we make this better?

Z boxes (boxen?)

T: ATACGGGCACATACCATACGAATATACAAA
Z_T: -1 0 1 0 0 0 0 1 0 4 0 1 0 0 5 0 1 0 0 1 3 0 4 0 1 0 1 1 1

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



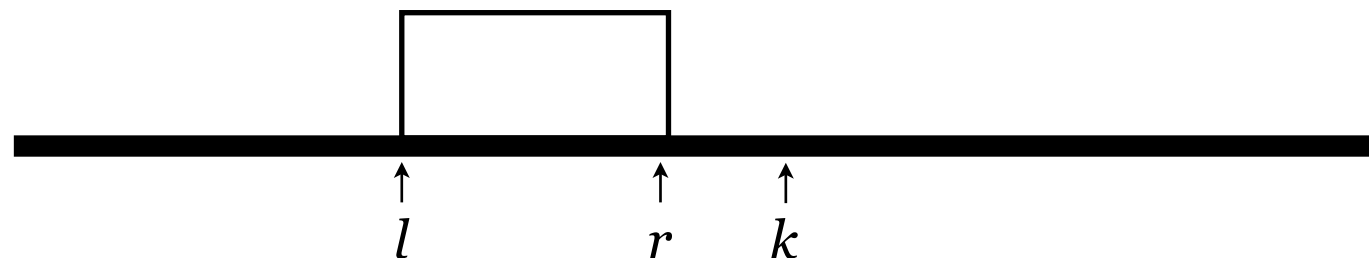
Def. *Z-box at i* is the substring starting at i and continuing to $i+Z_i-1$. This is the substring that matches the prefix. There is no Z-box at i if $Z_i = 0$.

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

Z Algorithm

- Input: $Z_2 \dots Z_{k-1}$, and the boundaries l, r of the rightmost Z-box found starting someplace in $2 \dots k-1$.
- 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 $Z_k > 0$

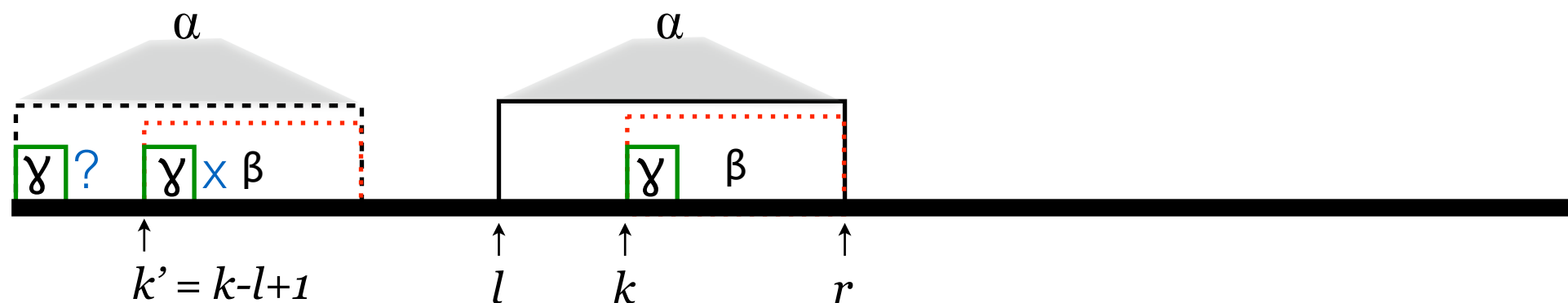
Z Algorithm

- Input: $Z_2 \dots Z_{k-1}$, and the boundaries l, r of the rightmost Z-box found starting someplace in $2 \dots k-1$.
- 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 γ that is a prefix of β is also a prefix of α , **but** the character occurring after the γ starting at k' is *not* the same as the character after the γ starting at the beginning of the string ... why?



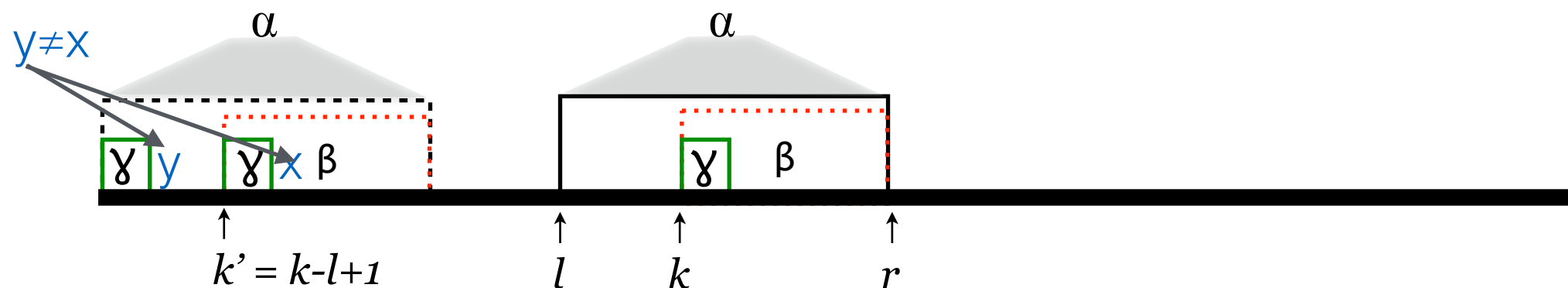
Z Algorithm

- Input: $Z_2 \dots Z_{k-1}$, and the boundaries l, r of the rightmost Z-box found starting someplace in $2 \dots k-1$.
- 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 y that is a prefix of β is also a prefix of α , **but** the character occurring after the y starting at k' is *not* the same as the character after the y starting at the beginning of the string ... **why?**



Z Algorithm

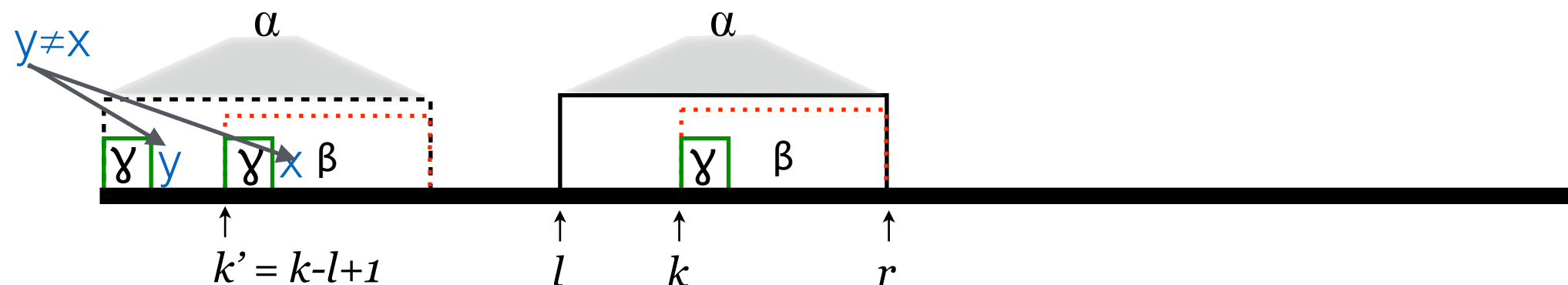
- Input: $Z_2 \dots Z_{k-1}$, and the boundaries l, r of the rightmost Z-box found starting someplace in $2 \dots k-1$.
- Output: Z_k , and updated l, r

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

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

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

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



Z Algorithm

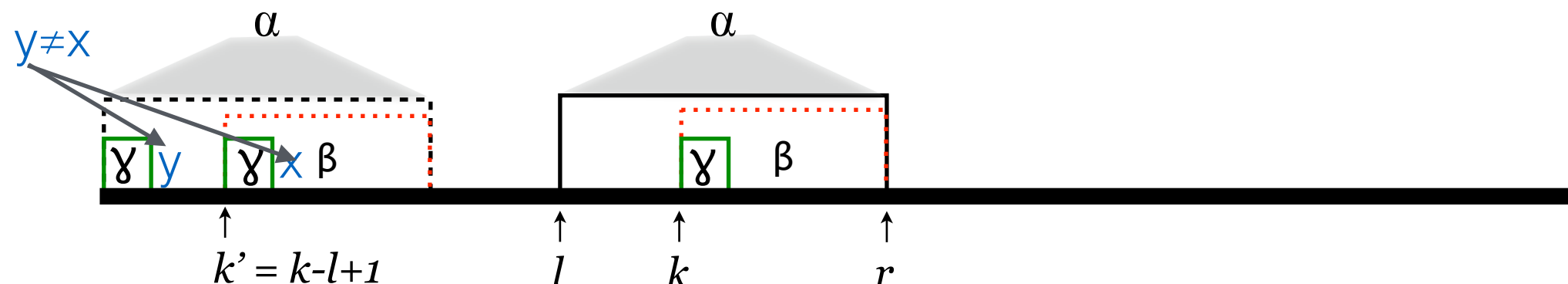
- Input: $Z_2 \dots Z_{k-1}$, and the boundaries l, r of the rightmost Z-box found starting someplace in $2 \dots k-1$.
- Output: Z_k , and updated l, r

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

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

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

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



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

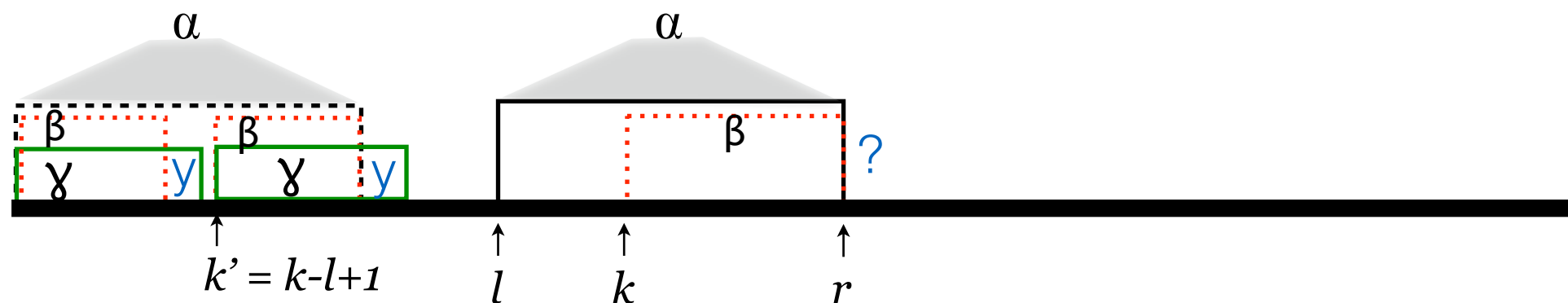
Z Algorithm

- Input: $Z_2 \dots Z_{k-1}$, and the boundaries l, r of the rightmost Z-box found starting someplace in $2 \dots k-1$.
- 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 y that starts at k' matches the y that starts at the beginning of T , **but**, it cannot (completely) match the substring starting at $k \dots$ why?



Z Algorithm

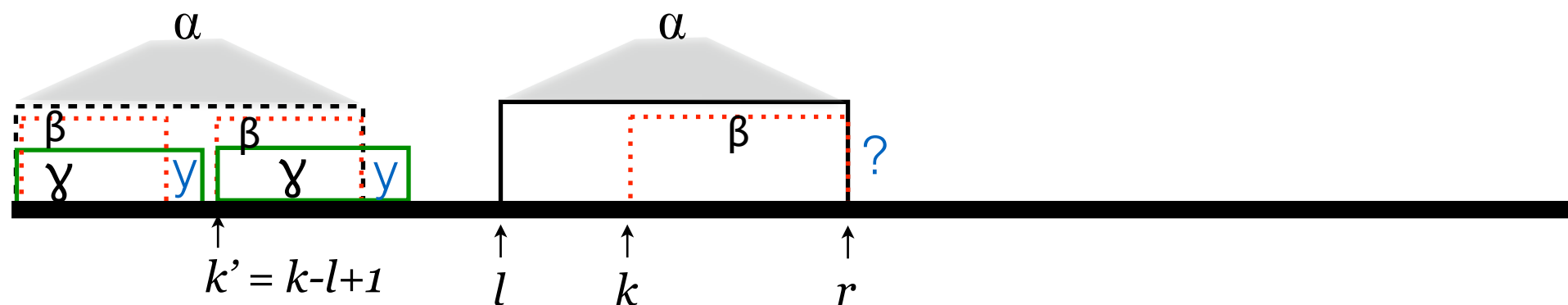
- Input: $Z_2 \dots Z_{k-1}$, and the boundaries l, r of the rightmost Z-box found starting someplace in $2 \dots k-1$.
- Output: Z_k , and updated l, r

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

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

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

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



Z Algorithm

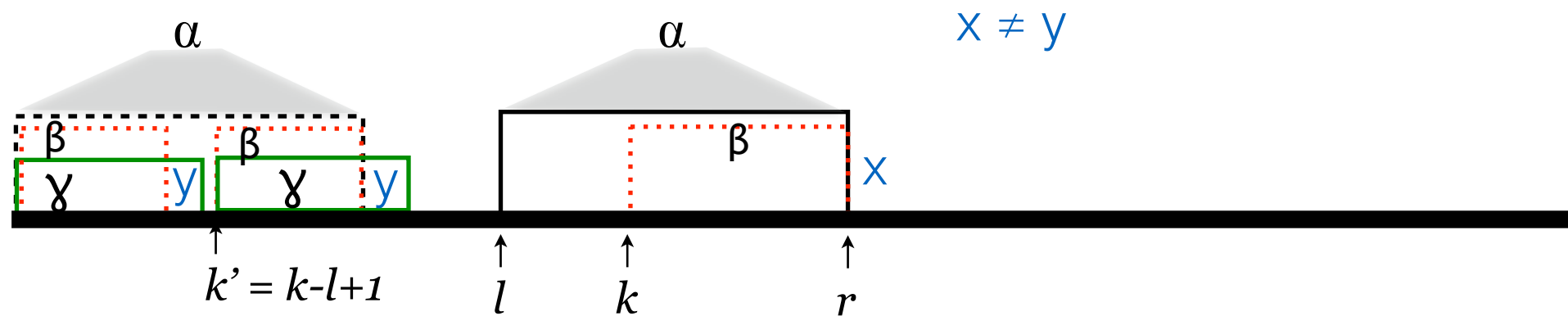
- Input: $Z_2 \dots Z_{k-1}$, and the boundaries l, r of the rightmost Z-box found starting someplace in $2 \dots k-1$.
- Output: Z_k , and updated l, r

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

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

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

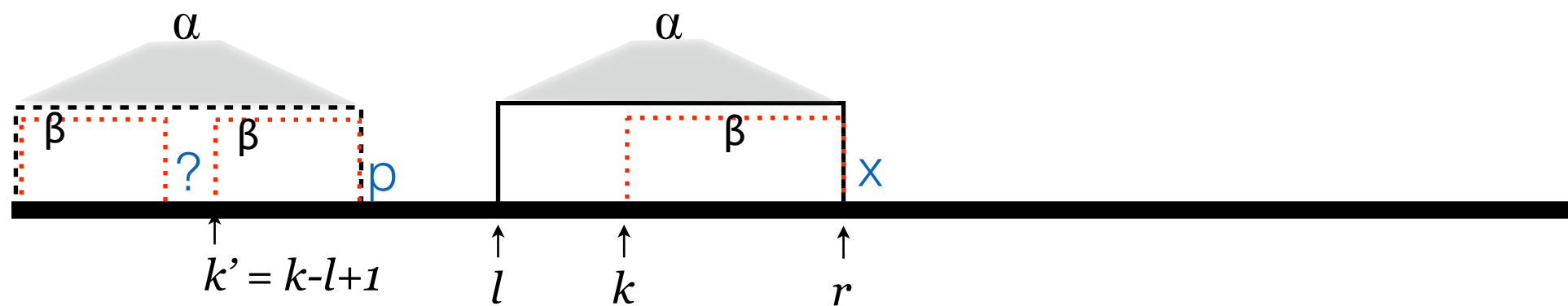
Z Algorithm

- Input: $Z_2 \dots Z_{k-1}$, and the boundaries l, r of the rightmost Z-box found starting someplace in $2 \dots k-1$.
- 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?



Z Algorithm

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

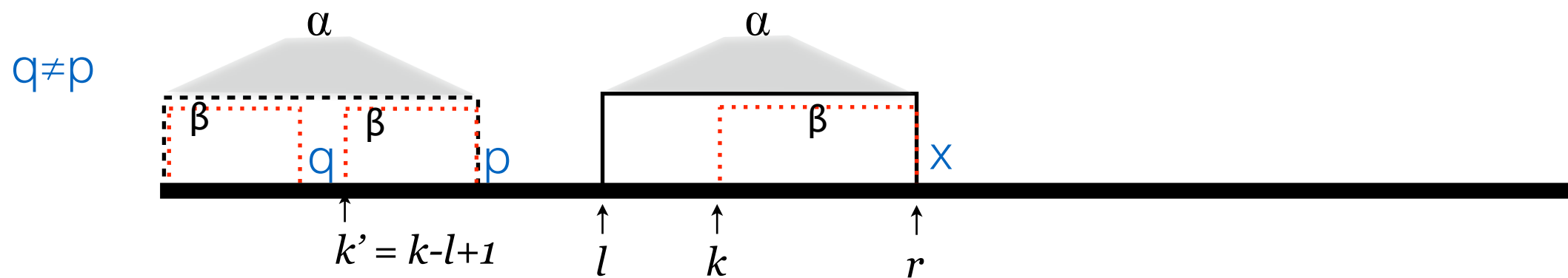
If $k \leq r$, this is the situation:

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 $> |\beta|$

What do we know about $x \dots x \neq p$. Is $x = q$?



Z Algorithm

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

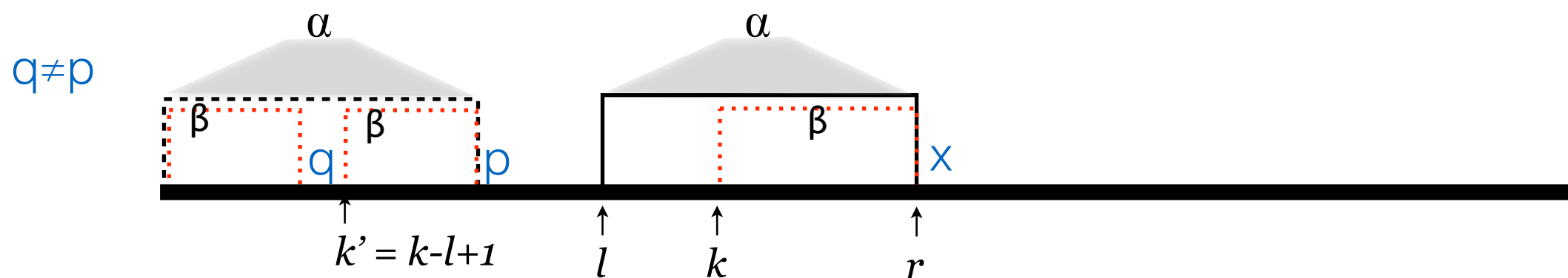
If $k \leq r$, this is the situation:

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 $> |\beta|$

What do we know about $x \dots x \neq p$. Is $x = q$? We don't know! Must check.



Explicitly compare after r to set Z_k . $l = k$, $r =$ point where comparison failed

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 $O(|S|)$ time:
 - only match characters covered by a Z-box once, so there are $O(|S|)$ matches.
 - every iteration contains at most one mismatch, so there are $O(|S|)$ 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.

Another algorithm

The key idea here will be exploiting redundancies (i.e. self-similarities) in the pattern **P**.

Say, we have:

T = CGAGACGAGAACGAGACGAGATCCCTCTAA

P = CGAGACGAGAT

CGAGACGAGACCGAGACGAGATCCCTCTAA
IIIIIIIIIX
CGAGACGAGAT

rather than shift **P** by 1 position, we can *skip* by a larger amount:

CGAGACGAGACCGAGACGAGATCCCTCTAA
CGAGACGAGAT

Next possible
occ. could start
here

But we know that
occ. would match
up until here

Knuth-Morris-Pratt Algorithm

Knuth, Donald E., James H. **Morris**, Jr, and Vaughan R. **Pratt**. "Fast pattern matching in strings." SIAM journal on computing 6.2 (1977): 323-350.

The Knuth-Morris-Pratt (KMP) algorithm provides an elegant approach to exploiting this intuition, allowing us to determine the optimal “skips”

Recall the following definitions:

String **s** is a **prefix/suffix** of **t** if **t** = **su/us** — if neither **s** nor **u** are ϵ , then **s** is a **proper prefix/suffix** of **t**

Knuth-Morris-Pratt Algorithm

Main idea: Build a *partial match* table, **pm**, that tells us, for each proper suffix of **P[0:q]**, the length of the longest match between this suffix and a proper prefix of **P[0:q]**.

In words, **pm[q]** is the number for which **P[0:pm[q]]** is the longest proper prefix of **P** that is also a proper suffix of **P[0:q]**

P	C	G	A	G	A	C	G	A	G	A	T
q	0	1	2	3	4	5	6	7	8	9	10
pm[q]	0	0	0	0	0	1	2	3	4	5	0

Knuth-Morris-Pratt Algorithm

CGAGACGAGAT

00000123450

The algorithm progresses as follows, assuming that $P[0:q-1]$ matches $T[i-q-1:i-1]$:

If $P[q] = T[i]$, then if $q < m$ we extend the length of the match, otherwise we've found a match and set $q = pm[q-1]$

Else $P[q] \neq T[i]$, then if $q = 0$ we increment i , otherwise we shift the pattern by $pm[q-1]$, and set $q = pm[q-1]$

Knuth-Morris-Pratt Algorithm

CGAGACGAGAT

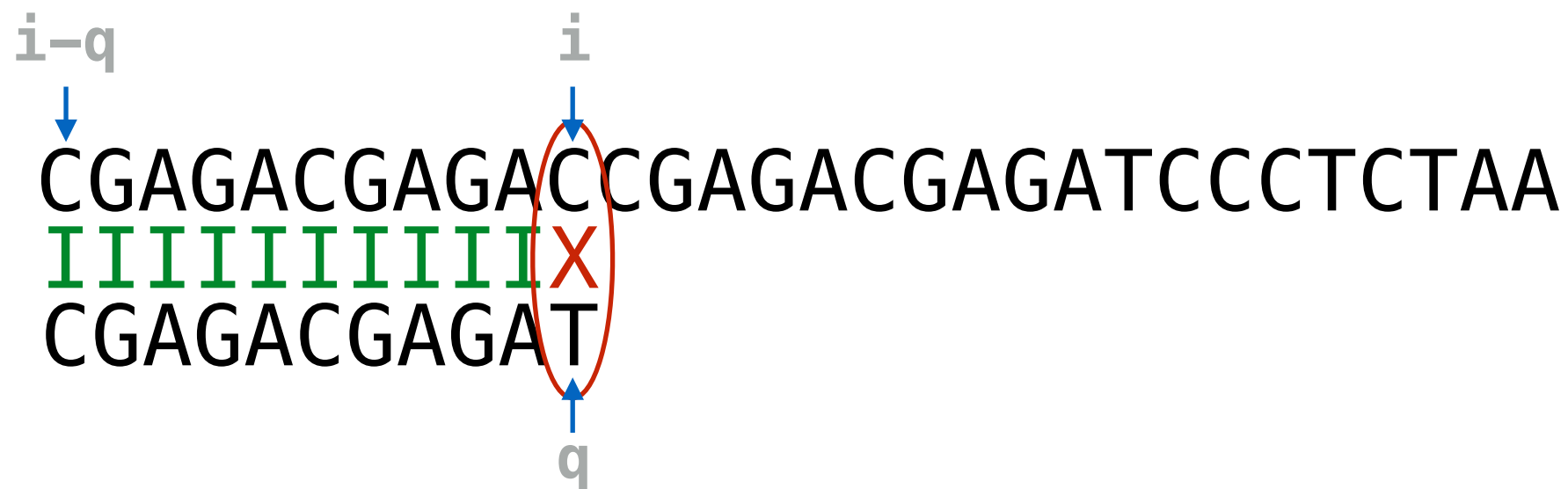
00000123450



Knuth-Morris-Pratt Algorithm

CGAGACGAGAT

00000123450

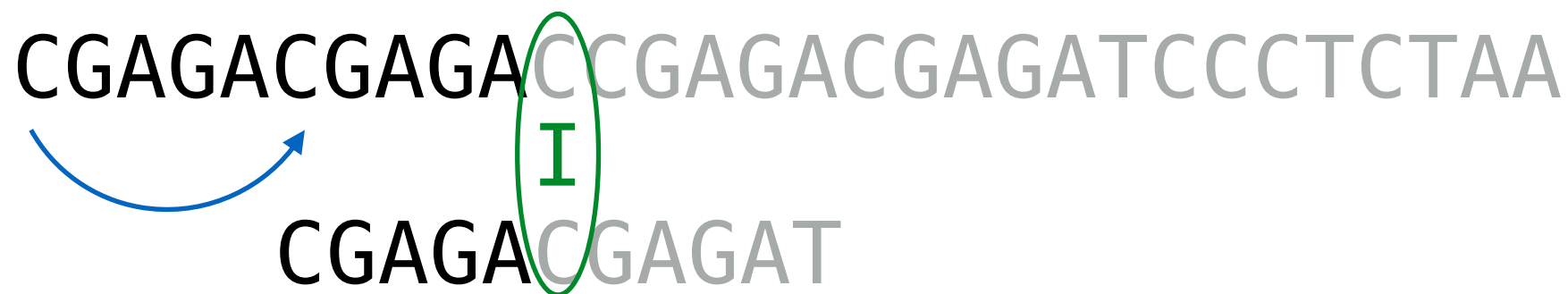


$T[i=10] \neq P[q=10]$, so we shift the pattern to the right by $pm[9] = 5$ and set $q = pm[q-1]$

Knuth-Morris-Pratt Algorithm



$T[i=10] \neq P[q=10]$, so we shift the pattern to the right by $pm[9] = 5$, setting $q = pm[q-1] = 5$



Even though we shift by 5, we actually skip even more character comparisons because we begin comparing the shifted pattern at position $q = 5$

```
def kmp(P,T):  
    n = len(T)  
    m = len(P)  
    matches = []  
    pi = partialMatchTable(P)  
    q = 0  
    i = 0  
    while i < n:  
        if P[q] == T[i]:  
            q += 1  
            i += 1  
            if q == m:  
                matches.append(i-q)  
                q = pi[q-1]  
        else:  
            if q == 0:  
                i += 1  
            else:  
                q = pi[q-1]  
    return matches
```

Running Time

Each pass through the outer loop either increments i or shifts the pattern to the right.

Both of these events can occur at most n times, and so, the loop, in total, can execute at most $2n = O(n)$ times.

Assuming pm is precomputed, each event takes $O(1)$ time.

Computing pm takes $O(m)$ time — we'll see that next

KMP runs in $O(n+m)$ time

Computing the Partial Match Table


```
def partialMatchTable(p):  
    m = len(p)  
    pm = [0] * m  
    k = 0  
    for q in range(1, m):  
        while k > 0 and p[k] != p[q]:  
            k = pm[k - 1]  
        if p[k] == p[q]:  
            k = k + 1  
        pm[q] = k  
    return pm
```

The key to the linearity of `partialMatchTable()` is that we always use `pm[0:i]` to compute `pm[i+1]`


```

def partialMatchTable(p):
    m = len(p)
    pm = [0] * m
    k = 0
    for q in range(1, m):
        while k > 0 and p[k] != p[q]:
            k = pm[k - 1]
        if p[k] == p[q]:
            k = k + 1
        pm[q] = k
    return pm

```



loop start: $m = 11$ $k = 0$ $q = 2$


loop end: $m = 11$ $k = 0$ $q = 2$

P	C	G	A	G	A	C	G	A	G	A	T
q	0	1	2								
pm[q]	0	0	0								

```

def partialMatchTable(p):
    m = len(p)
    pm = [0] * m
    k = 0
    for q in range(1, m):
        while k > 0 and p[k] != p[q]:
            k = pm[k - 1]
        if p[k] == p[q]:
            k = k + 1
        pm[q] = k
    return pm

```



loop start: $m = 11$ $k = 0$ $q = 3$


loop end: $m = 11$ $k = 0$ $q = 3$

P	C	G	A	G	A	C	G	A	G	A	T
q	0	1	2	3							
pm[q]	0	0	0	0							

```

def partialMatchTable(p):
    m = len(p)
    pm = [0] * m
    k = 0
    for q in range(1, m):
        while k > 0 and p[k] != p[q]:
            k = pm[k - 1]
        if p[k] == p[q]:
            k = k + 1
        pm[q] = k
    return pm

```



loop start: $m = 11$ $k = 0$ $q = 4$


loop end: $m = 11$ $k = 0$ $q = 4$

P	C	G	A	G	A	C	G	A	G	A	T
q	0	1	2	3	4						
pm[q]	0	0	0	0	0						

```

def partialMatchTable(p):
    m = len(p)
    pm = [0] * m
    k = 0
    for q in range(1, m):
        while k > 0 and p[k] != p[q]:
            k = pm[k - 1]
        if p[k] == p[q]:
            k = k + 1
        pm[q] = k
    return pm

```



loop start: $m = 11$ $k = 0$ $q = 5$


loop end: $m = 11$ $k = \mathbf{1}$ $q = 5$

P	C	G	A	G	A	C	G	A	G	A	T
q	0	1	2	3	4	5					
pm[q]	0	0	0	0	0	1					

```

def partialMatchTable(p):
    m = len(p)
    pm = [0] * m
    k = 0
    for q in range(1, m):
        while k > 0 and p[k] != p[q]:
            k = pm[k - 1]
        if p[k] == p[q]:
            k = k + 1
        pm[q] = k
    return pm

```



loop start: $m = 11$ $k = 1$ $q = 6$


loop end: $m = 11$ $k = \mathbf{2}$ $q = \mathbf{6}$

P	C	G	A	G	A	C	G	A	G	A	T
q	0	1	2	3	4	5	6				
pm[q]	0	0	0	0	0	1	2				

```

def partialMatchTable(p):
    m = len(p)
    pm = [0] * m
    k = 0
    for q in range(1, m):
        while k > 0 and p[k] != p[q]:
            k = pm[k - 1]
        if p[k] == p[q]:
            k = k + 1
        pm[q] = k
    return pm

```



loop start: $m = 11$ $k = 2$ $q = 7$


loop end: $m = 11$ $k = \mathbf{3}$ $q = \mathbf{7}$

P	C	G	A	G	A	C	G	A	G	A	T
q	0	1	2	3	4	5	6	7			
pm[q]	0	0	0	0	0	1	2	3			

```

def partialMatchTable(p):
    m = len(p)
    pm = [0] * m
    k = 0
    for q in range(1, m):
        while k > 0 and p[k] != p[q]:
            k = pm[k - 1]
        if p[k] == p[q]:
            k = k + 1
        pm[q] = k
    return pm

```



loop start: $m = 11$ $k = 3$ $q = 8$


loop end: $m = 11$ $k = \mathbf{4}$ $q = 8$

P	C	G	A	G	A	C	G	A	G	A	T
q	0	1	2	3	4	5	6	7	8		
pm[q]	0	0	0	0	0	1	2	3	4		

```

def partialMatchTable(p):
    m = len(p)
    pm = [0] * m
    k = 0
    for q in range(1, m):
        while k > 0 and p[k] != p[q]:
            k = pm[k - 1]
        if p[k] == p[q]:
            k = k + 1
        pm[q] = k
    return pm

```



loop start: $m = 11$ $k = 4$ $q = 9$

loop end: $m = 11$ $k = \mathbf{5}$ $q = 9$

P	C	G	A	G	A	C	G	A	G	A	T
q	0	1	2	3	4	5	6	7	8	9	
pm[q]	0	0	0	0	0	1	2	3	4	5	


```
def partialMatchTable(p):
    m = len(p)
    pm = [0] * m
    k = 0
    for q in range(1, m):
        while k > 0 and p[k] != p[q]:
            k = pm[k - 1]
        if p[k] == p[q]:
            k = k + 1
        pm[q] = k
    return pm
```

When this happens,
 $k = pm[5-1] = 0$, so
 the while loop executes
 once.

loop start: $m = 11$ $k = 5$ $q = 10$

loop end: $m = 11$ $k = \mathbf{0}$ $q = 10$

P	C	G	A	G	A	C	G	A	G	A	T
q	0	1	2	3	4	5	6	7	8	9	10
pm[q]	0	0	0	0	0	1	2	3	4	5	0

Summary

Despite our ability to solve general pairwise alignment, exact matching is still important

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

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

Dan Gusfield is awesome!

Knuth, Morris & Pratt are awesome!

Next time, we'll see how to do even better by pre-processing the *text*.