CSE 549: Computational Biology

Exact String Matching

Stony Brook University

slides (w/*) courtesy of Carl Kingsford
or (w/+) Ben Langmead

Why Exact Matching?

We’ll motivate the next few lectures with the problem of “read alignment” — finding the occurrences of sequencing reads in a target genome.

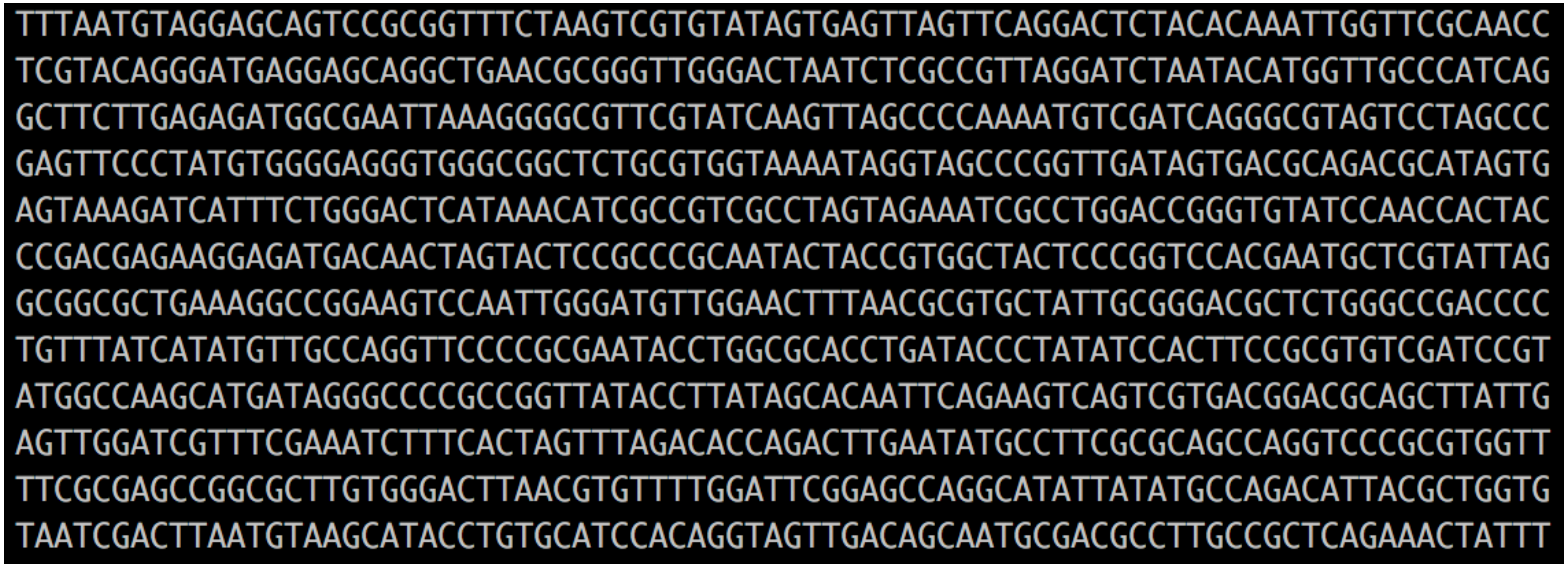
However, read alignment is an instance of the *approximate* pattern matching problem.

Nonetheless, we’ll use the *simpler* problem of *exact* pattern matching, and generalize the solutions later.

Exact String Matching Problem

Today, we'll talk about exact matching algorithms that are **quadratic** and **linear**. Then we'll start talking about ***much*** faster approaches, but they require pre-processing the reference.

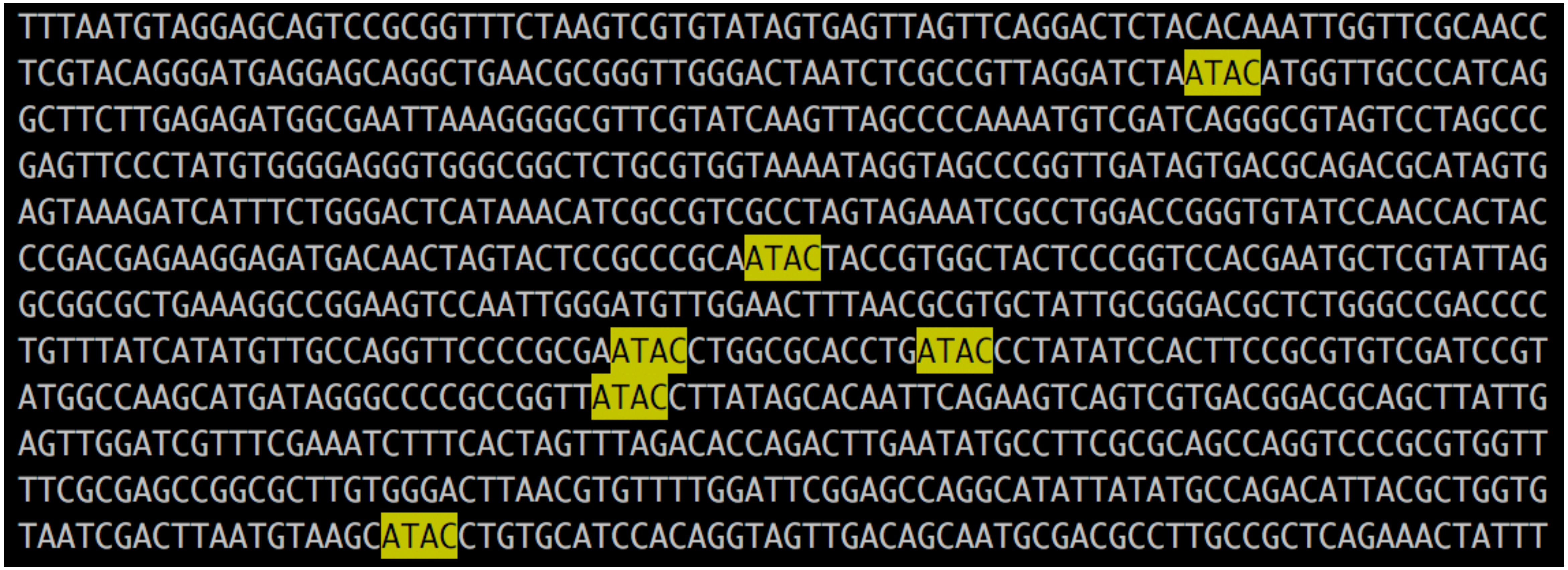
Finding needles in a haystack



```
TTTAATGTAGGAGCAGTCCGCGGTTTCTAAGTCGTGTATAGTGAGTTAGTTCAGGACTCTACACAAATTGGTTCGCAACC
TCGTACAGGGATGAGGAGCAGGCTGAACGCGGGTTGGGACTAATCTCGCCGTTAGGATCTAATACATGGTTGCCCATCAG
GCTTCTTGAGAGATGGCGAATTAAAGGGGCGTTCGTATCAAGTTAGCCCCAAAATGTCGATCAGGGCGTAGTCCTAGCCC
GAGTTCCCTATGTGGGGAGGGTGGGCGGCTCTGCGTGGTAAAATAGGTAGCCCGGTTGATAGTGACGCAGACGCATAGTG
AGTAAAGATCATTTCTGGGACTCATAAACATCGCCGTCGCCTAGTAGAAATCGCCTGGACCGGGTGTATCCAACCACTAC
CCGACGAGAAGGAGATGACAACTAGTACTCCGCCCGCAATACTACCGTGGCTACTCCCGGTCCACGAATGCTCGTATTAG
GCGGCGCTGAAAGGCCGGAAGTCCAATTGGGATGTTGGAACTTTAACGCGTGCTATTGCGGGACGCTCTGGGCCGACCCC
TGTTTATCATATGTTGCCAGGTTCCCCGCGAATACCTGGCGCACCTGATACCCTATATCCACTTCCGCGTGTCGATCCGT
ATGGCCAAGCATGATAGGGCCCCGCCGGTTATACCTTATAGCACAATTCAGAAGTCAGTCGTGACGGACGCAGCTTATTG
AGTTGGATCGTTTCGAAATCTTTCACTAGTTTAGACACCAGACTTGAATATGCCTTCGCGCAGCCAGGTCCCGCGTGGTT
TTCGCGAGCCGGCGCTTGTGGGACTTAACGTGTTTTGGATTCGGAGCCAGGCATATTATATGCCAGACATTACGCTGGTG
TAATCGACTTAATGTAAGCATACCTGTGCATCCACAGGTAGTTGACAGCAATGCGACGCCTTGCCGCTCAGAAACTATTT
```

Where does “ATAC” occur?

Finding needles in a haystack



```
TTTAATGTAGGAGCAGTCCGCGGTTTCTAAGTCGTGTATAGTGAGTTAGTTCAGGACTCTACACAAATTGGTTCGCAACC
TCGTACAGGGATGAGGAGCAGGCTGAACGCGGGTTGGGACTAATCTCGCCGTTAGGATCTAATACATGGTTGCCCATCAG
GCTTCTTGAGAGATGGCGAATTAAAGGGGCGTTCGTATCAAGTTAGCCCCAAAATGTCGATCAGGGCGTAGTCCTAGCCC
GAGTTCCCTATGTGGGGAGGGTGGGCGGCTCTGCGTGGTAAAATAGGTAGCCCGGTTGATAGTGACGCAGACGCATAGTG
AGTAAAGATCATTTCTGGGACTCATAAACATCGCCGTCGCCTAGTAGAAATCGCCTGGACCGGGTGTATCCAACCACTAC
CCGACGAGAAGGAGATGACAACTAGTACTCCGCCCGCAATACTACCGTGGCTACTCCCGGTCCACGAATGCTCGTATTAG
GCGGCGCTGAAAGGCCGGAAGTCCAATTGGGATGTTGGAACTTTAACGCGTGCTATTGCGGGACGCTCTGGGCCGACCCC
TGTTTATCATATGTTGCCAGGTTCCCCGCGAATACCTGGCGCACCTGATACCCTATATCCACTTCCGCGTGTCGATCCGT
ATGGCCAAGCATGATAGGGCCCCGCCGGTTATACCTTATAGCACAATTCAGAAGTCAGTCGTGACGGACGCAGCTTATTG
AGTTGGATCGTTTCGAAATCTTTCACTAGTTTAGACACCAGACTTGAATATGCCTTCGCGCAGCCAGGTCCCGCGTGGTT
TTCGCGAGCCGGCGCTTGTGGGACTTAACGTGTTTTGGATTCGGAGCCAGGCATATTATATGCCAGACATTACGCTGGTG
TAATCGACTTAATGTAAGCATACCTGTGCATCCACAGGTAGTTGACAGCAATGCGACGCCTTGCCGCTCAGAAACTATTT
```

The Language of Strings

A string **s** is a finite sequence of characters

|**s**| denotes the length of the string — the number of characters in the sequence.

A string is defined *over* an alphabet, Σ

$\Sigma_{DNA} = \{A,T,C,G\}$
$\Sigma_{RNA} = \{A,U,C,G\}$
$\Sigma_{AminoAcid} = \{A, R, N, D, C, E, Q, G, H, I, L, K, M, F, P, S, T, W, Y, V\}$

The empty string is denoted ϵ — $|\epsilon| = 0$

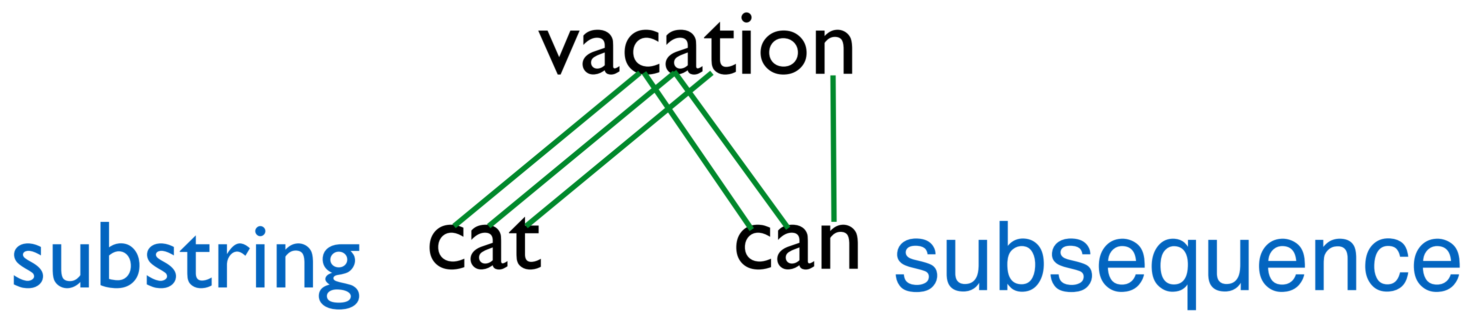
+

The Language of Strings

Given two strings **s,t** over the same alphabet Σ, we denote the concatenation as **st** — this is the sequence of **s** followed by the sequence of **t**

String **s** is a substring of **t** if there exist two (potentially empty) strings **u** and **v** such that **t = usv**

String **s** is a subsequence of **t** if the characters of **s** appear in order (but not necessarily consecutively) in **t**



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**

+

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 = ATACATACCCATATA|CGAG|GCATACATGG|CGAG|TGTGC
P = CGAG           |    |          |    |
                   |CGAG|          |CGAG|
```

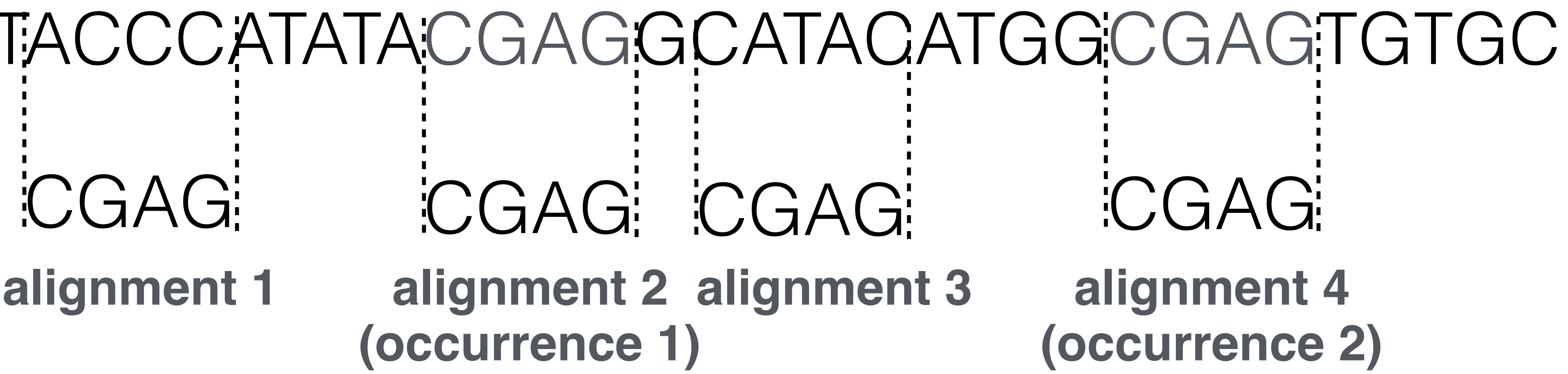
Occurrences vs. Alignments

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



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 xrange(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

O(N)

```
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
```

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?

Another algorithm

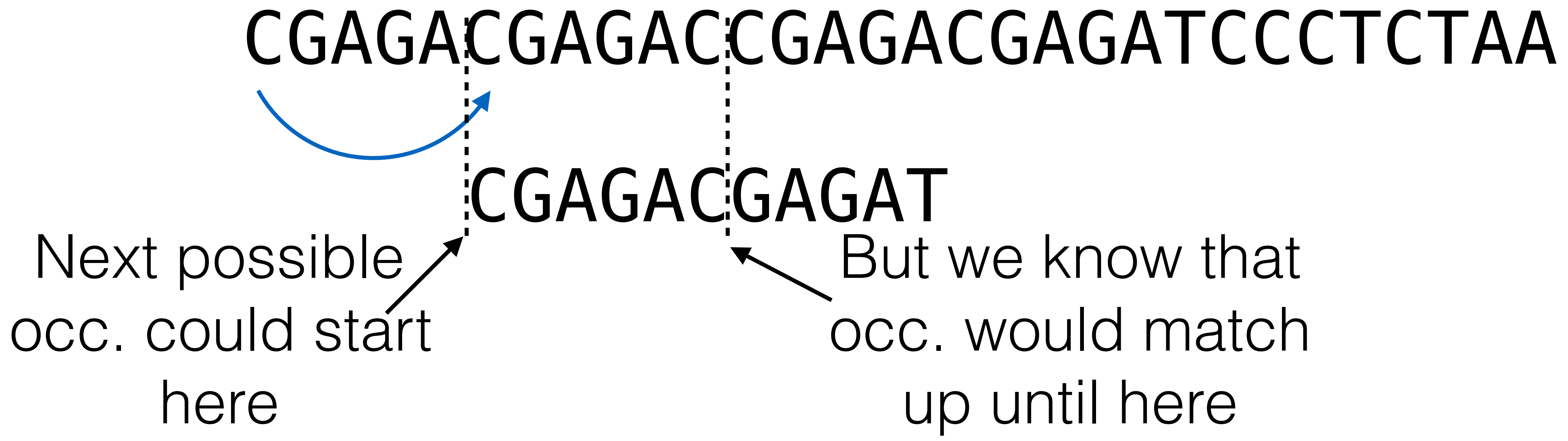
The key idea here will be exploiting redundancies (i.e. self-similarities) in the pattern **P**.

Say, we have:

```
T = CGAGACGAGAACGAGACGAGATCCCTCTAA
P = CGAGACGAGAT
```

```
CGAGACGAGACCGAGACGAGATCCCTCTAA
IIIIIIIIIIX
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

```
 P: CGAGACGAGAT
pm: 00000123450
```

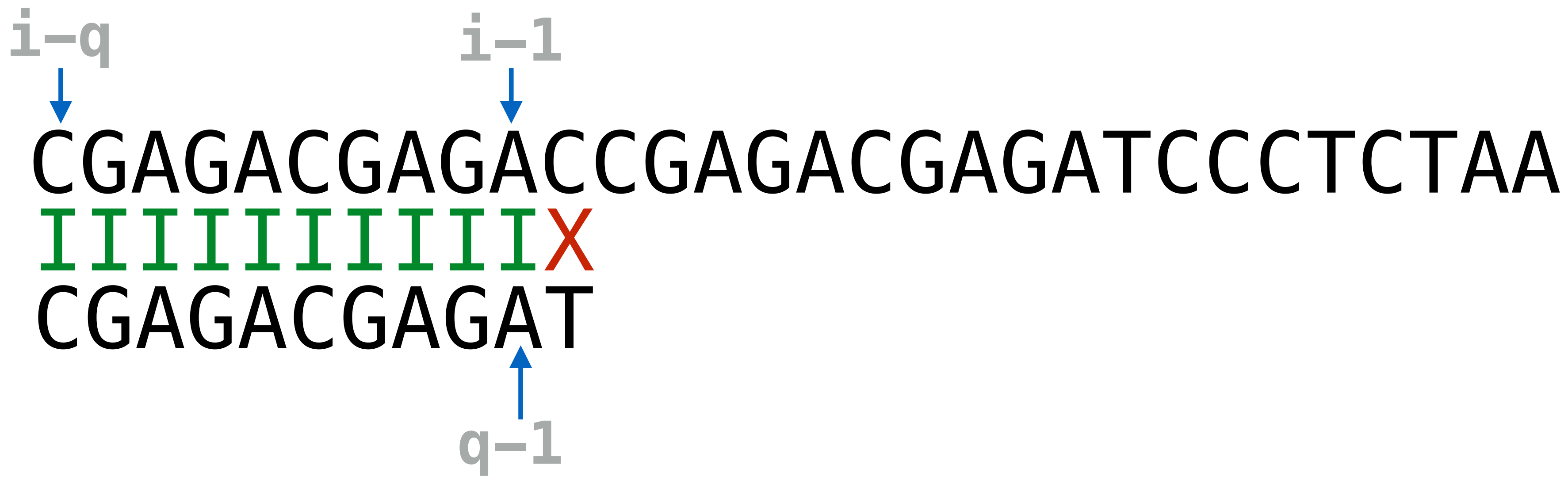
The algorithm progresses as follows, assuming that P[0:q-1] matches T[i-q, i-1]:

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] ≠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

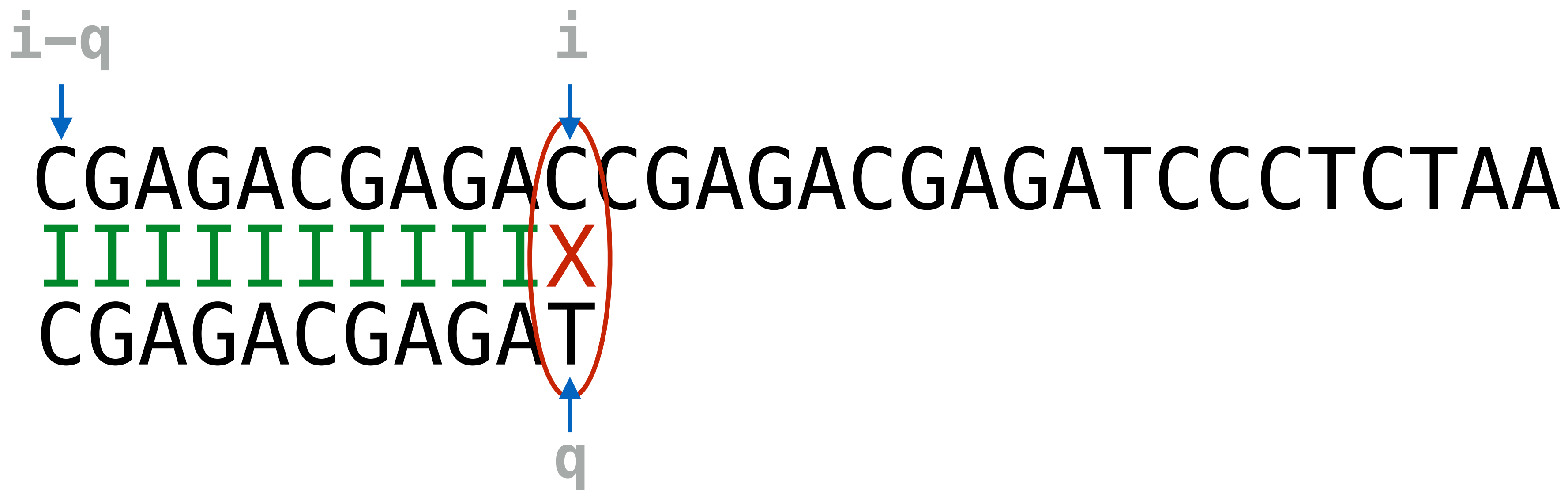
```
 P: CGAGACGAGAT
pm: 00000123450
```



```
i-q              i-1
CGAGACGAGACCGAGACGAGATCCCTCTAA
IIIIIIIIIIX
CGAGACGAGAT
                 q-1
```

Knuth-Morris-Pratt Algorithm

```
 P: CGAGACGAGAT
pm: 00000123450
```



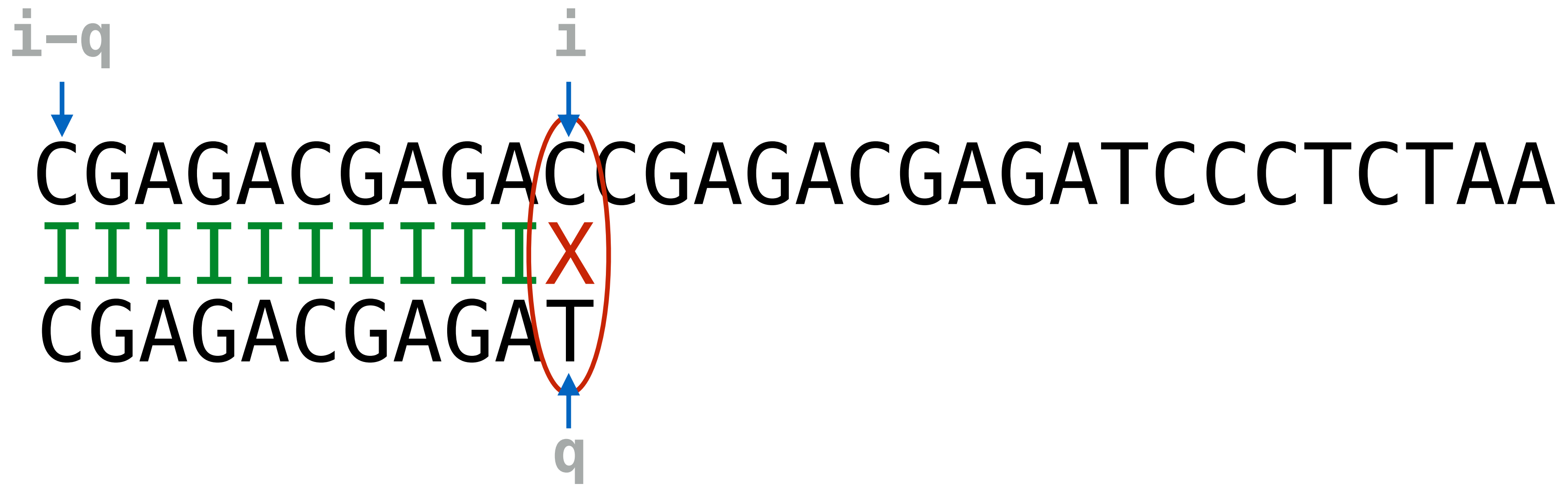
```
CGAGACGAGACCGAGACGAGATCCCTCTAA
IIIIIIIIIIX
CGAGACGAGAT
```

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

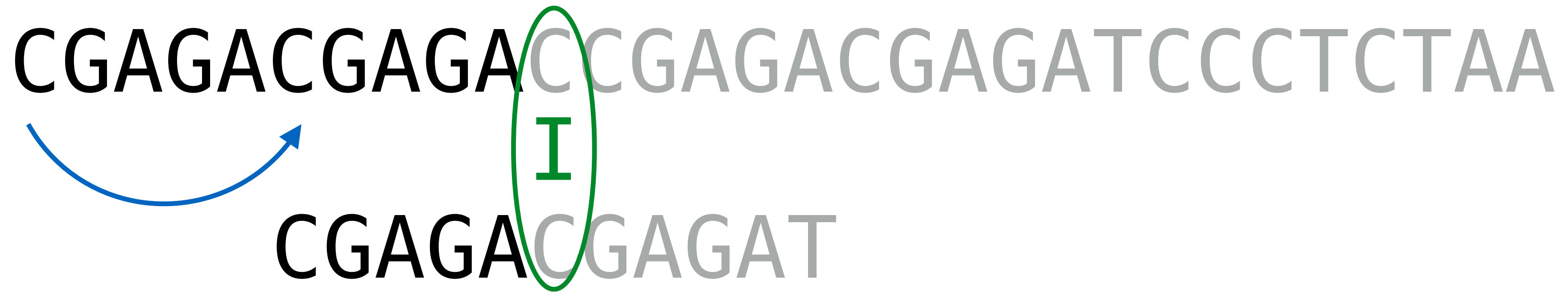
Knuth-Morris-Pratt Algorithm

P: CGAGACGAGAT

pm: 00000123450



T[i=10] ≠ 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

```python
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

analysis following: http://www.cs.ubc.ca/~hoos/cpsc445/Handouts/kmp.pdf

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 = 1

loop end: m = 11   k = 0   q = 1

P	C	G	A	G	A	C	G	A	G	A	T
q	0	1									
pm[q]	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 = 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: $m = 11$ $k =$ **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 = **2**   q = **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 = **3** q = **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 = **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 = **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 = **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)

Knuth, Morris & Pratt are awesome!

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