CSE 549: Computational Biology

Exact String Matching

slides (w/*) courtesy of Carl Kingsford
or (w/+) Ben Langmead
Why Exact Matching?

Already discussed how to perform optimal (semi)-global/local alignment, why worry about the simpler problem of exact string matching?

As we saw, our alignment algorithms scale as O(nm). When n \approx 10^9 and m \approx 10^2 this becomes intractable (especially when we 10 of millions of strings of length \sim m)
Why Exact Matching?

Even ignoring, e.g memory access, say filling in each matrix cell takes \( C = 10 \) CPU cycles.

\[
N = 10^9 \quad \quad M = 10^2 \quad \quad R = 10^7
\]

order of genome \quad \quad order of read length \quad \quad order of # of reads

\[\text{# of ops} \approx N \times M \times R \times C = 10^{19}\]

\[\text{ops/sec} \approx 3 \times 10^9 \quad (3\text{GHz CPU})\]

\[\text{# ops / (ops/sec)} = \text{secs} \approx \frac{10^{19}}{(3 \times 10^9)} = (1/3) \times 10^{10}\]
Why Exact Matching?

Even ignoring, e.g memory access, say filling in each matrix cell takes \( C = 10 \) CPU cycles.

\[
\begin{align*}
N &= 10^9 \\
M &= 10^2 \\
R &= 10^7 \\
\text{order of genome} &\quad \text{order of read length} & \text{order of # of reads}
\end{align*}
\]

\[
\# \text{ of ops } \approx N \times M \times R \times C = 10^{19}
\]

\[
\text{ops/sec } \approx 3 \times 10^9 \quad \text{(3GHz CPU)}
\]

\[
\# \text{ ops } / (\text{ops/sec}) = \text{secs } \approx 10^{19} / (3 \times 10^9) = (1/3) \times 10^{10}
\]

\(~106 \text{ Years! (for a relatively small 10M read dataset)~}\)
So, nobody does a naive optimal alignment to map reads

Typical strategy (many variants):

• 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
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 $P$.

$T =$ ATACATACCCATATA CGAG GCATACATGG CGAG TG TGTGC

$P =$ CGAG

CGAG

CGAG

CGAG
An alignment of $P$ to $T$ is a correspondence between $T$ and a substring of $P$ — all occurrences are alignment but not all alignments are occurrences.

$T = \text{ATACATACCCATATACGAGGCGATACATGGCGAGTGTGTC}$

$P = \text{CGAG}$

alignment 1

alignment 2 (occurrence 1)

alignment 3

alignment 4 (occurrence 2)
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.

```python
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

```python
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 xrange(N - M + 1):
        if P == T[i:i+M]:
            occs.append(i)
    return occs

O(N) * O(M) = O(NM) time
A naive algorithm

Best scenario for naive:

T: GAGAGGGAGTTATATATGAATAGAGATAGAGACGAG

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: CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

P: CCCC

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:

$$T = \text{CGAGACGAGAACGAGACGAGATCCCTCTAA}$$

$$P = \text{CGAGACGAGAT}$$

CGAGACGAGACCGAGACGAGATCCCTCTAA

\[\text{IIIIIIIIIIIX}\]

CGAGACGAGAT

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

CGAGACGAGACCGAGACGAGATCCCTCTAA

Next possible occ. could start here

But we know that occ. would match up until here

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 $\epsilon$, 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 \( P[0:q] \).
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] ≠ 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

i-q-1  i-1

CGAGACGAGACCGAGACGAGATCCCTCTAA
IIIIIIIIIIIX
CGAGACGAGAT

q-1
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

CGAGACGAGAT
00000123450

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

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

Computing $p_m$ takes $O(m)$ time — we’ll see that next

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

Computing the Partial Match Table

```python
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

<table>
<thead>
<tr>
<th>P</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pm[q]</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</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

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

<table>
<thead>
<tr>
<th>P</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pm[q]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</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

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

<table>
<thead>
<tr>
<th>P</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pm[q]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</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

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

<table>
<thead>
<tr>
<th>P</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| pm[q] | 0 | 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

<table>
<thead>
<tr>
<th>P</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pm[q]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```python
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

```
<table>
<thead>
<tr>
<th>P</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pm[q]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
```python
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
```

### Example:

**Loop Start:**
- `m = 11`
- `k = 2`
- `q = 7`

**Loop End:**
- `m = 11`
- `k = 3`
- `q = 7`

---

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>pm[q]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
```python
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
```

<table>
<thead>
<tr>
<th>P</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pm[q]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

loop start: m = 11  k = 3  q = 8
loop end: m = 11  k = 4  q = 8
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

<table>
<thead>
<tr>
<th>P</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>pm[q]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</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

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

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>pm[q]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>
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 preprocessing the text.