More on the Motif Problem

- Exhaustive Search and Median String are both exact algorithms.
- They always find the optimal solution, though they may be too slow to perform practical tasks.
- Many algorithms sacrifice optimal solution for speed.
Some Motif Finding Programs

- **CONSENSUS**
  Hertz, Stromo (1989)

- **GibbsDNA**
  Lawrence et al (1993)

- **MEME**
  Bailey, Elkan (1995)

- **RandomProjections**
  Buhler, Tompa (2002)

- **MULTIPROFILER**
  Keich, Pevzner (2002)

- **MITRA**
  Eskin, Pevzner (2002)

- **Pattern Branching**
CONSENSUS: Greedy Motif Search

- Find two closest l-mers in sequences 1 and 2 and forms a $2 \times l$ alignment matrix with $\text{Score}(s, 2, \text{DNA})$

- At each of the following $t$-2 iterations CONSENSUS finds a “best” $l$-mer in sequence $i$ from the perspective of the already constructed $(i-1) \times l$ alignment matrix for the first $(i-1)$ sequences.

- In other words, it finds an $l$-mer in sequence $i$ maximizing $\text{Score}(s, i, \text{DNA})$

under the assumption that the first $(i-1)$ $l$-mers have been already chosen.

- CONSENSUS sacrifices optimal solution for speed: in fact the bulk of the time is actually spent locating the first 2 $l$-mers.
EXACT STRING MATCHING
The problem of String Matching

Given a string ‘t’, the problem of string matching deals with finding whether a pattern ‘p’ occurs in ‘t’ and if ‘p’ does occur then returning position in ‘t’ where ‘p’ occurs.
Brute force \(O(mn))\)

\[
n \leftarrow |t|
m \leftarrow |p|
i \leq 1
\]

while \(i < n\)
\[
    \text{if } p == t[i, i+m-1] \\
    \quad \text{return } i;
\]
else
\[
    i = i + 1;
\]
## SimpleStringSearch

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

|   | Y    | Y    | Y    | Y    | Y    | N    |
**SimpleStringSearch**

<table>
<thead>
<tr>
<th>(t[0])</th>
<th>(t[1])</th>
<th>(t[2])</th>
<th>(t[3])</th>
<th>(t[4])</th>
<th>(t[5])</th>
<th>(t[6])</th>
<th>(t[7])</th>
<th>(t[8])</th>
<th>(t[9])</th>
<th>(t[10])</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(p[0])</th>
<th>(p[1])</th>
<th>(p[2])</th>
<th>(p[3])</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>
SimpleStringSearch

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

N
# SimpleStringSearch

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

N
### SimpleStringSearch

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

N
SimpleStringSearch

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

N
## SimpleStringSearch

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

Y Y Y Y Y
Straightforward string searching

- **Worst case:**
  - Pattern string always matches completely except for last character
  - Example: search for `XXXXXY` in target string of `XXXXXXXXXXXXXXXXXXXXXXXX`
  - Outer loop executed once for every character in target string
  - Inner loop executed once for every character in pattern
  - $O(mn)$, where $m = |p|$ and $n = |t|$

- Okay if patterns are short, but better algorithms exist
Knuth-Morris-Pratt

- $O(m+n)$

- Key idea:
  - if pattern fails to match, slide pattern to right by as many boxes as possible without permitting a match to go unnoticed
Knuth-Morris-Pratt’s algorithm compares the pattern to the text in left-to-right, but shifts the pattern more intelligently than the brute-force algorithm.

When a mismatch occurs, what is the most we can shift the pattern so as to avoid redundant comparisons?

Answer: the largest prefix of $P[0..j]$ that is a suffix of $P[1..j]$
KMP Failure Function

- Knuth-Morris-Pratt’s algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself.

- The **failure function** $F(j)$ is defined as the size of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$.

- Knuth-Morris-Pratt’s algorithm modifies the brute-force algorithm so that if a mismatch occurs at $P[j] \neq T[i]$ we set $j \leftarrow F(j - 1)$.

<table>
<thead>
<tr>
<th>$j$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P[j]$</td>
<td>$a$</td>
<td>$b$</td>
<td>$a$</td>
<td>$a$</td>
<td>$b$</td>
<td>$a$</td>
</tr>
<tr>
<td>$F(j)$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
The KMP Algorithm

- The failure function can be represented by an array and can be computed in $O(m)$ time
- At each iteration of the while-loop, either
  - $i$ increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j - 1) < j$)
- Hence, there are no more than $2n$ iterations of the while-loop
- Thus, KMP’s algorithm runs in optimal time $O(m + n)$

Algorithm $KMPMatch(T, P)$

1. $F \leftarrow failureFunction(P)$
2. $i \leftarrow 0$
3. $j \leftarrow 0$
4. while $i < n$
   - if $T[i] = P[j]$
     - if $j = m - 1$
       - return $i - j$ { match }
     - else
       - $i \leftarrow i + 1$
       - $j \leftarrow j + 1$
   - else
     - if $j > 0$
       - $j \leftarrow F[j - 1]$
     - else
       - $i \leftarrow i + 1$
5. return $-1$ { no match }
Computing the Failure Function

- The failure function can be represented by an array and can be computed in $O(m)$ time.
- The construction is similar to the KMP algorithm itself.
- At each iteration of the while-loop, either
  - $i$ increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j - 1) < j$).
- Hence, there are no more than $2m$ iterations of the while-loop.

**Algorithm** $\text{failureFunction}(P)$

```plaintext
F[0] \leftarrow 0
i \leftarrow 1
j \leftarrow 0

while i < m
    if $P[i] = P[j]$  
        {we have matched $j + 1$ chars}
            $F[i] \leftarrow i + 1$
            $i \leftarrow i + 1$
            $j \leftarrow j + 1$
    else if $j > 0$ then
        {use failure function to shift $P$}
            $j \leftarrow F[j - 1]$
    else
        $F[i] \leftarrow 0$ { no match }
        $i \leftarrow i + 1$
```
Example

<table>
<thead>
<tr>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>P[j]</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>F(j)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

```
[Diagram showing the example with sequences and indices]
```
The Boyer-Moore Algorithm

- Similar to KMP in that:
  - Pattern compared against target
  - On mismatch, move as far to right as possible

- Different from KMP in that:
  - Compare the patterns from right to left instead of left to right

- Does that make a difference?
  - Yes – much faster on long targets; many characters in target string are never examined at all
There is no E in the pattern: thus the pattern can’t match if any characters lie under t[3]. So, move four boxes to the right.
Again, no match. But there is a B in the pattern. So move two boxes to the right.
Boyer-Moore example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

Y Y Y Y Y Y
**Boyer-Moore : another example**

<table>
<thead>
<tr>
<th>$t[k]$</th>
<th>$t[k+1]$</th>
<th>...</th>
<th>$t[k+i]$</th>
<th>$t[k+m-1]$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>...</td>
<td>C</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$p[0]$</th>
<th>$p[1]$</th>
<th>...</th>
<th>$p[i-1]$</th>
<th>$p[i]$</th>
<th>$p[i+1]$</th>
<th>...</th>
<th>$p[m-1]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>E</td>
<td>...</td>
<td>S</td>
<td>D</td>
<td>E</td>
<td>...</td>
<td>R</td>
</tr>
</tbody>
</table>

Problem: determine $d$, the number of boxes that the pattern can be moved to the right.

$d$ should be smallest integer such that $t[k+m-1] = p[m-1-d]$, $t[k+m-2] = p[m-2-d]$, ... $t[k+i] = p[i-d]$
The Boyer-Moore Algorithm

- We said:
  - $d$ should be smallest integer such that:
    - $T[k+m-1] = p[m-1-d]$
    - $T[k+m-2] = p[m-2-d]$
    - $T[k+i] = p[i-d]$
  - Reminder:
    - $k = \text{starting index in target string}$
    - $m = \text{length of pattern}$
    - $i = \text{index of mismatch in pattern string}$
  - Problem: statement is valid only for $d \leq i$
    - Need to ensure that we don’t “fall off” the left edge of the pattern
Boyer-Moore: another example

If $c == W$, then $d$ should be 3

If $c == R$, then $d$ should be 7
Bad Character Rule

Suppose that $P_1$ is aligned to $T_s$ now, and we perform a pair-wise comparing between text $T$ and pattern $P$ from right to left. Assume that the first mismatch occurs when comparing $T_{s+j-1}$ with $P_j$.

Since $T_{s+j-1} \neq P_j$, we move the pattern $P$ to the right such that the largest position $c$ in the left of $P_j$ is equal to $T_{s+j-1}$. We can shift the pattern at least $(j-c)$ positions right.
Rule 2-1: Character Matching Rule
(A Special Version of Rule 2)

- Bad character rule uses Rule 2-1 (Character Matching Rule).
- For any character $x$ in $T$, find the nearest $x$ in $P$ which is to the left of $x$ in $T$. 

![Diagram showing the matching process between T and P]
Case 1. If there is a x in P to the left of T, move P so that the two x’s match.
Case 2: If no such a $x$ exists in $P$, move $P$ to the right of $x$
Ex: Suppose that P1 is aligned to T6 now. We compare pairwise between T and P from right to left. Since T16,17 = P11,12 = “CA” and T15 = “G” ≠ P10 = “T”. Therefore, we find the rightmost position c=7 in the left of P10 in P such that Pc is equal to “G” and we can move the window at least (10-7=3) positions.
Good Suffix Rule 1

- If a mismatch occurs in $T_{s+j-1}$, we match $T_{s+j-1}$ with $P_{j'-m+j}$, where $j' \ (m-j+1 \leq j' < m)$ is the largest position such that
  
  1. $P_{j+1,m}$ is a suffix of $P_{1,j'}$.
  2. $P_{j'-(m-j)} \neq P_j$.

- We can move the window at least $(m-j')$ position(s).
Rule 2: The Substring Matching Rule

- For any substring $u$ in $T$, find a nearest $u$ in $P$ which is to the left of it. If such a $u$ in $P$ exists, move $P$;
Suppose that P1 is aligned to T6 now. We compare pair-wise between P and T from right to left. Since T16,17 = “CA” = P11,12 and T15 = “A” ≠ P10 = “T”. We find the substring “CA” in the left of P10 in P such that “CA” is the suffix of P1,6 and the left character to this substring “CA” in P is not equal to P10 = “T”. Therefore, we can move the window at least m-j’ (12-6=6) positions right.
Good Suffix Rule 2

Good Suffix Rule 2 is used only when Good Suffix Rule 1 cannot be used. That is, t does not appear in $P(1, j)$. Thus, t is unique in $P$.

If a mismatch occurs in $T_{s+j-1}$, we match $T_{s+m-j'}$ with $P_1$, where $j' (1 \leq j' \leq m-j)$ is the largest position such that $P_{1,j'}$ is a suffix of $P_{j+1,m}$.

P.S. : $t'$ is suffix of substring t.
Rule 3-1: Unique Substring Rule

- The substring \( u \) appears in \( P \) exactly once.
- If the substring \( u \) matches with \( T_{i,j} \), no matter whether a mismatch occurs in some position of \( P \) or not, we can slide the window by \( l \).

The string \( s \) is the longest prefix of \( P \) which equals to a suffix of \( u \).
Rule 1: The Suffix to Prefix Rule

For a window to have any chance to match a pattern, in some way, there must be a suffix of the window which is equal to a prefix of the pattern.
Rule 1: The Suffix to Prefix Rule

- Note that the above rule also uses Rule 1.
- It should also be noted that the unique substring is the shorter and the more right-sided the better.
- A short \( u \) guarantees a short (or even empty) \( s \) which is desirable.
Ex: Suppose that $P_1$ is aligned to $T_6$ now. We compare pair-wise between $P$ and $T$ from right to left. Since $T_{12} \neq P_7$ and there is no substring $P_{8,12}$ in left of $P_8$ to exactly match $T_{13,17}$. We find a longest suffix “AATC” of substring $T_{13,17}$, the longest suffix is also prefix of $P$. We shift the window such that the last character of prefix substring to match the last character of the suffix substring. Therefore, we can shift at least 12-4=8 positions.
Let \( B(a) \) be the rightmost position of \( a \) in \( P \). The function will be used for applying **bad character rule**.

We can move our pattern right at least \( j-B(T_{s+j-1}) \) position by above \( B \) function.

Move at least \( 10-B(G) = 10 \) positions
Let $G_s(j)$ be the largest number of shifts by good suffix rule when a mismatch occurs for comparing $P_j$ with some character in $T$. 
• \(gs_1(j)\) be the largest \(k\) such that \(P_{j+1,m}\) is a suffix of \(P_{1,k}\) and \(P_{k-m+j} \neq P_j\), where \(m-j+1 \leq k < m\); 0 if there is no such \(k\).

(gs\(_1\) is for Good Suffix Rule 1)

• \(gs_2(j)\) be the largest \(k\) such that \(P_{1,k}\) is a suffix of \(P_{j+1,m}\), where \(1 \leq k \leq m-j\); 0 if there is no such \(k\).

(gs\(_2\) is for Good Suffix Rule 2.)

• \(Gs(j) = m - \max\{gs_1, gs_2\}\), if \(j = m\), \(Gs(j)=1\).

\[
\begin{array}{cccccccccccc}
  j & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
  gs_1 & 0 & 0 & 0 & 0 & 0 & 0 & 9 & 0 & 0 & 6 & 1 & 0 \\
  gs_2 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 1 & 1 & 1 & 0 \\
  Gs & 8 & 8 & 8 & 8 & 8 & 8 & 3 & 8 & 11 & 6 & 11 & 1 \\
\end{array}
\]

\(gs_1(7)=9\)

\(\therefore P_{8,12}\) is a suffix of \(P_{1,9}\) and \(P_4 \neq P_7\)

\(gs_2(7)=4\)

\(\therefore P_{1,4}\) is a suffix of \(P_{8,12}\)
Time Complexity

- The preprocessing phase in $O(m+\Sigma)$ complexity
- If you are searching for ALL matches, worst case:
  - $O(mn)$ when $P$ is in $T$
    - $T=AAAAAAAAAAAAA; \ P=AAAA$
  - $O(m+n)$ when $P$ is not in $T$