Pattern Matching (Exact Matching)

Young-Rae Cho
Assistant Professor
Department of Computer Science
Baylor University

Overview

- Pattern Matching
  - Exhaustive Search
  - DFA Algorithm
  - KMP Algorithm
- Multiple Pattern Matching
  - AC Algorithm
  - Weiner’s Algorithm
- Approximate Pattern Matching
- Pattern Finding
- Approximate Pattern Finding (Sequence Motif Finding)
Pattern Matching

- **Definition**
  - Given a text (string), finding all occurrences of a pattern (substring)
  - Given a DNA, RNA, or protein sequence, finding all occurrences of a specific repeat

- **Examples**
  - `ATGGTC TAGGTC CTAGGTGC`

- **Applications**
  1. Homolog search in BLAST
  2. Sequence motif search
     - Repeats (substrings, patterns) often represent sequence motifs
     - Functional domains are often associated with repeats
     - Evolutionary path can be traced by repeats

Terminology

- **Prefix**
  - `S[1 .. j]` is a prefix of a string `S[1 .. n]` where `j ≤ n`
  - `X` is a prefix of `Y` if `X·Z=Y` for some string `Z`

- **Suffix**
  - `S[i .. n]` is a suffix of a string `S[1 .. n]` where `1 ≤ i`
  - `X` is a suffix of `Y` if `Z·X=Y` for some string `Z`

- **Substring**
  - A string of consecutive letters from `S`
  - `S[i .. j]` is a substring of a string `S[1 .. n]` where `1 ≤ i` and `j ≤ n`
  - A substring of `S` is a prefix of a suffix of `S`

- **Empty String**
  - `S[i .. j]` is an empty string where `i > j`
Properties

- Proper Prefix, Proper Suffix, Proper Substring
  - The proper prefix, suffix, or substring of a string S is a prefix, suffix, or substring that is not the empty string nor S itself

- Main Properties
  - Reflexivity (But, not for proper prefix, proper suffix, proper substring)
  - Anti-symmetry (But, not for proper prefix, proper suffix, proper substring)
  - Transitivity

- Other Properties
  - If X is a suffix of Y, then X·Z is a suffix of Y·Z for some string Z
  - If X is a suffix of Z, Y is a suffix of Z, and |X|≤|Y|, then X is a suffix of Y

Formulation of Pattern Matching Problem

- Goal
  - Finding all occurrences of a substring (length-m) in a string (length-n)

- Input
  - A substring \( P = p_1 \cdot p_2 \cdot \ldots \cdot p_m \) and a string \( T = t_1 \cdot t_2 \cdot \ldots \cdot t_n \)

- Output
  - All positions \( 1 \leq i \leq (n-m+1) \) such that the substring of \( T \) starting at \( i \) matches \( P \)
Naïve Approach

- Algorithm
  - Exhaustive search

```
NaiveMatching(T, P)
n ← length(T)
m ← length(P)
for i ← 1 to n − m + 1
  if P[1..m] = T[i..(i + m)]
    then print i
```

- Example
  - $T = CTGCATC$
  - $P = GCAT$

<table>
<thead>
<tr>
<th>CTGCATC</th>
<th>CTGCATC</th>
<th>CTCCATC</th>
<th>CTGCATC</th>
<th>CTGCATC</th>
<th>CTGCATC</th>
<th>CTGCATC</th>
<th>CTGCATC</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCAT</td>
<td>GCAT</td>
<td>GCAT</td>
<td>GCAT</td>
<td>GCAT</td>
<td>GCAT</td>
<td>GCAT</td>
<td>GCAT</td>
</tr>
</tbody>
</table>

- Runtime ?

Deterministic Finite Automata (1)

- Definition
  - A device involving states and transitions among them in response to inputs

- Finite Automata $M = (Q, q_0, A, \Sigma, \delta)$
  - $Q$: a finite set of states
  - $q_0$: a start state
  - $A$: a set of accepting states
  - $\Sigma$: a finite set of input characters
  - $\delta$: transition function from $Q \times \Sigma$ into $Q$

- Acceptance
  - Automaton accepts the input string if it ends up in an accepting state
  - Automaton rejects the input string if it doesn’t end up in an accepting state
Deterministic Finite Automata (2)

- **Applications**
  - Lexical analysis of a compiler
  - System for verifying the correctness of circuits or protocols

- **Examples**
  - Verifying inputs
  - Constructing automata

- **Pattern Matching with DFA**
  1. Constructs an automaton for the substring (pattern) \( P \)
  2. Searches \( P \) by reading the string (text) \( T \) on the automaton

Constructing DFA (1)

- **Suffix Function**
  - Suffix function \( \sigma(X) \) for \( P \): a mapping to the length of the longest prefix of \( P \) that is a suffix of \( X \)
  - e.g., \( P = \text{"abc"}, \sigma(\text{"cbaca"}) = ?, \sigma(\text{"ccab"}) = ? \)

- **Process**
  - Given a substring (pattern) \( P \) with length \( m \)
  - Makes the set of states \( Q = \{0,1, ..., m\} \), with the state 0 as \( q_0 \), and the state \( m \) as the only accepting state
  - Defines the transition function \( \delta \) as
    \[
    \delta(q, a) = \sigma(P[1..q] \cdot a)
    \]
Constructing DFA (2)

- Example
  - $P=\text{"ababaa"}, \Sigma=\{a,b\}$

- Runtime?

Searching Pattern using DFA (1)

- Process
  - Given an input string $T$ having the letters in $\Sigma$
  - Starts at the state $q_0$
  - Reads the string $T$, character by character, changing state after each character read

- Pattern Matching
  - Automaton finds the substring $P$ from $T$ if it reaches an accepting state

- Example
  - $P=\text{"ababaa"}, \Sigma=\{a,b\}$
  - $T=\text{"aababaababaababaa"}
Searching Pattern using DFA (2)

- Algorithm

\[
\text{AutomataMatching}(T, P, M) \\
\begin{align*}
u & \leftarrow \text{length}(T) \\
m & \leftarrow \text{length}(P) \\
q & \leftarrow q_0 \\
\text{for } i \leftarrow 1 \text{ to } n \\
q & \leftarrow \delta(q, T[i]) \\
\text{if } q \in A \\
& \text{ then print } (i - m + 1)
\end{align*}
\]

- Runtime ?

- Total Runtime of Pattern Matching with DFA ?

Pattern Shifting

- Backgrounds

\[
\begin{array}{cccccccccccc}
& b & a & c & b & a & b & a & b & a & b & c & b & a & b \\
T & s & & & & & & & & & & & & \\
\begin{array}{cccc}
& a & b & a & b & a & c & a & \end{array}
\]

\[
\begin{array}{cccccccccccc}
& b & a & c & b & a & b & a & b & a & b & a & b & c & b & a & b \\
T & s'=s+2 & & & & & & & & & & & & \\
\begin{array}{cccc}
& a & b & a & b & a & c & a & \end{array}
\]
\]

- \(T[1..n]\), \(P[1..m]\)
- Given \(P[1..q]\) (where \(q \leq m\)) matches \(T[(s+1)..(s+q)]\), what is the least shift \(s'\) (where \(s'>s\)) such that
  \[
P[1..k] = T[(s'+1)..(s'+k)] \text{ where } s'+k = s+q
\]
Prefix Function

- Prefix Function
  - Prefix function $\pi(q)$ for $P$: a mapping to the length of the longest prefix of $P$ that is a proper suffix of $P[1..q]$

**Example**

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P[i]$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>$\pi(i)$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$T$</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>c</th>
<th>c</th>
<th>a</th>
<th>b</th>
<th>b</th>
<th>a</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
</tbody>
</table>

$\pi(5) = 3$

$\pi(4) = 2$

$\pi(2) = 0$

Knuth-Morris-Pratt Algorithm (1)

**Algorithm**

```
KMP-MATCHING(T, P, $\pi$)
    n ← length(T)
    m ← length(P)
    q ← 0
    for $i ← 1$ to $n$
        while $q > 0$ and $P[q + 1] \neq T[i]$
            $q ← \pi[q]$
        if $P[q + 1] = T[i]$
            then $q ← q + 1$
        if $q = m$
            then print ($i - m + 1$) and $q ← \pi[q]$
```

**Runtime ?**
Knuth-Morris-Pratt Algorithm (2)

- Algorithm of Prefix Function

  ```
  PrefixFunction(P)
  m ← length(P)
  π[1] ← 0
  k ← 0
  for q ← 2 to m
    while k > 0 and P[k + 1] ≠ P[q]
      k ← π[k]
    if P[k + 1] = P[q]
      k ← k + 1
    π[q] ← k
  return π
  ```

- Runtime ?

- Total Runtime of KMP Algorithm ?

Overview

- Pattern Matching
  - Exhaustive Search
  - DFA Algorithm
  - KMP Algorithm

- Multiple Pattern Matching
  - AC Algorithm
  - Weiner’s Algorithm

- Approximate Pattern Matching

- Pattern Finding

- Approximate Pattern Finding (Sequence Motif Finding)
Multiple Pattern Matching

- Motivation
  - Finding matches of multiple patterns from a text at the same time
  - Finding all occurrences of multiple patterns at the same time in a DNA or protein sequence improves efficiency for homolog search

- Examples
  - ATGGTCTAGGTCCCTAGTGTC
  - P = { GGTC, CTAG, TGGT }

Formulation of Multiple Pattern Matching Problem

- Goal
  - Finding all occurrences of any in a set of substrings (length-$m$) in a string (length-$n$)

- Input
  - A set of $k$ substrings $P_1, P_2, \ldots, P_k$ and a string $T = t_1 \cdot t_2 \cdot \ldots \cdot t_n$

- Output
  - All positions $1 \leq i \leq n$ such that a substring of $T$ starting at $i$ matches $P_j$ where $1 \leq j \leq k$
Extension of Pattern Matching

- Extension of Naïve Approach
  - Naïve string matching $k$ times
  - Runtime ?

- Extension of Other String Matching Algorithms
  - KMP string matching $k$ times
  - Runtime ?

- Direction
  - Advanced data structure
  - Advanced algorithm

Prefix Tree (1)

- Prefix Tree
  - Data structure to manage a set of substrings (patterns), $P$
  - Each path from the root represents each pattern
  - Also called "keyword tree" or "trie"

- Features
  - Each edge is labeled with a character
  - Any two or more edges to child nodes from a parent node have different labels
  - Each node $v$ is labeled as the concatenation of edge labels on the path from the root to $v$ (the node label is denoted by $L(v)$)
  - For each $P_i \in P$, there is a node $v$ such that $L(v) = P_i$
  - $L(v)$ for any leaf node $v$ equals some $P_j$ where $P_j \in P$
Prefix Tree (2)

- Example
  - \( P = \{ TG, ATG, TCA, TGAC \} \)

- Runtime of Prefix Tree Construction?

Extension of Finite Automata (1)

- Multiple String Matching with Finite Automata (Aho-Corasick Algorithm)
  1. Constructs an automaton for the set of substrings (patterns), \( P \)
  2. Searches all substrings in the string (text) \( T \) by the automaton

- Finite Automata, \( M=(Q, q_0, A, \Sigma, \delta) \), on a Prefix Tree
  - \( Q \) : the set of nodes in the prefix tree
  - \( q_0 \) : the root in the prefix tree
  - \( A \) : the nodes marked in the prefix tree
  - \( \Sigma \) : the set of all distinct characters in \( P \)
  - \( \delta \) : transition functions
    - goto functions \( (g) \)
    - failure functions \( (f) \)
Extension of Finite Automata (2)

- Goto Function
  - \( g(q_i, a) \): a mapping to the state entered from the current state \( q_i \)
    - by matching the target character \( a \)
    - If the edge \((q_i, q_j)\) is labeled by \( a \), and \( q_i \) is a parent node of \( q_j \) in the
      prefix tree, then \( g(q_i, a) = q_j \)
    - Otherwise, \( g(q_i, a) = \emptyset \), except \( g(q_0, a) = q_0 \)

- Failure Function
  - \( f(q_i) = \pi(L(q_i)) \): a mapping to the state of the longest prefix of
    some pattern in \( P \), which is a proper suffix of \( L(q_i) \)

Example of Finite Automata

- Example
  - \( P = \{ TG, ATG, TCA, TGAC \} \)
Searching Multiple Patterns by FA (1)

- **Process**
  - Given an input string $T$ having the letters in $\Sigma$,
  - Starts at the state $q_0$,
  - Reads the string $T$, character by character, changing state after each character read.

- **Multiple String Matching**
  - Automaton finds a substring $P_j$ in $P$ from $T$ if it reaches the accepting state corresponding to $P_j$.

- **Example**
  - $P = \{ \text{TG, ATG, TCA, TGAC} \}$
  - $T = \text{ATCATGTGAC}$

Searching Multiple Patterns by FA (2)

- **Algorithm**

```
AC-MULTIPLE-MATCHING(T, \{P_1, P_2, \ldots, P_k\}, M)
    n ← length(T)
    for j = 1 to k
        m_j ← length(P_j)
        q ← q_0
        for i = 1 to n
            while g(q, T[i]) = ∅
                q ← f(q)
            q ← g(q, T[i])
            if q = a_j ∈ A
                then print (i - m_j + 1)
```

- **Runtime ?**
Constructing Finite Automata (1)

- Process
  - Constructs the prefix tree for $P$
    - all nodes in the prefix tree $\rightarrow Q$
    - the root node $\rightarrow q_0$
  - Marks all accepting states for $A$
  - Makes goto function for each state
  - Makes failure function for each state as $f(q) = \pi(L(q))$

Constructing Finite Automata (2)

- Algorithm of Failure Function

```plaintext
FAILURE_FUNCTION(M, g)
Q $\leftarrow$ empty queue
for $a \in \Sigma$
  if $g(q_0, a) = q \neq q_0$
    $f(q) \leftarrow 0$ and enqueue(q, Q)
while $Q \neq \emptyset$
  $v$ $\leftarrow$ dequeue(Q)
  for $a \in \Sigma$
    if $g(v, a) = u \neq \emptyset$
      enqueue(u, Q) and $v \leftarrow f(v)$
    while $g(v, a) = \emptyset$
      $v \leftarrow f(v)$
    $f(u) \leftarrow g(v, a)$
return $f$
```

- Runtime ?

- Total Runtime of AC Algorithm ?
Suffix Tree (1)

- **Suffix Tree**
  - Data structure to manage a string (text), $T$
  - Each path from the root represents each suffix of $T$
  - Also called "collapsed keyword tree"

- **Features**
  - Each edge is labeled with a string (a substring of $T$)
  - All internal nodes have at least two outgoing edges
    ⇒ Similar to prefix trees, but edges that form a linear path are collapsed
  - Leaf nodes are labeled with the index of the pattern (starting position)

Suffix Tree (2)

- **Examples**
  - $T = \text{ATCATG}$
    - T
    - TCATG
    - CATG
    - ATG
    - TG
    - G

- **Runtime of Suffix Tree**
  - Construction ?
    - Naïve approach

(a) Keyword tree  
(b) Suffix tree
Constructing Suffix Tree (1)

- Weiner’s Algorithm
  - Linear-time suffix tree construction algorithm
- Substring Function
  - Substring function \( \theta(i) \) for \( T \): a mapping to the position and length of the substring of \( T[(i+1) \ldots n] \) that matches the longest prefix of \( T[i \ldots n] \)

<table>
<thead>
<tr>
<th>( i )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T[i] )</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>G</td>
</tr>
<tr>
<td>( \theta(i) )</td>
<td>4/2</td>
<td>5/1</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
<td>0/0</td>
</tr>
</tbody>
</table>

Constructing Suffix Tree (2)

- Process
  - Read each character in \( T \) backwards
  - Attach the suffix \( T[i \ldots n] \) to the nodes labeled with the position of \( \theta(i) \)
  - Converting the edge with the length of \( \theta(i) \), and adding new branches
  - Example, \( T = ATCATG \)
Multiple Matching with Suffix Tree

- Process
  - Build a suffix tree for $T$
  - Thread each pattern $P_i$ where $1 \leq i \leq k$ through the suffix tree
  - If threading is complete, output all labels of leaf nodes

- Example of Threading
  - $T = \text{"ATGCATA\text{CATG}"}$
  - $P_i = \text{"ATG"}$

- Runtime?

Overview

- Pattern Matching
  - Exhaustive Search
  - DFA Algorithm
  - KMP Algorithm

- Multiple Pattern Matching
  - AC Algorithm
  - Weiner’s Algorithm

- Approximate Pattern Matching
- Pattern Finding
- Approximate Pattern Finding (Sequence Motif Finding)
Exact Matching vs. Inexact Matching

- **Exact Matching**
  
  \[ T = \text{agcctccgatgtaagtcatcgatgtaactattcgatgacctccctattacatcttacgatgtcataca} \]
  \[ P = \text{"cgatgt"} \]
  
  \[ T = \text{agcctccgatgtaagtcatcgatgtaactattcgatgacctccctattacatcttacgatgtcataca} \]

- **Inexact Matching (Approximate Matching)**

  \[ T = \text{agcctccgatgtaagtcatcgatgtaactattcgatgacctccctattacatcttacgatgtcataca} \]
  \[ P = \text{"cgatgt"} \]
  \[ P = \text{"cgatgt"} \]
  \[ P = \text{"cgatgt"} \]

Formulation of Approximate Matching Problem

- **Goal**
  - Finding all approximate occurrences of a substring (length-\(m\)) in a string (length-\(n\))

- **Input**
  - A substring \(P = p_1, \ldots, p_m\), a string \(T = t_1, \ldots, t_n\), and the maximum number mismatches, \(k\)

- **Output**
  - All positions \(1 \leq i \leq (n-m+1)\) such that \(P\) and the substring of \(T\) starting at \(i\) have at most \(k\) mismatches
Naïve Approach

- **Algorithm**
  - Exhaustive search

  ```
  APPROXIMATEMATCHING(T, P, k)
  n ← length(T)
  m ← length(P)
  for i ← 1 to n - m + 1
    mismatch ← 0
    for j ← 1 to m
      if T[i + j - 1] ≠ P[j]
        then mismatch ← mismatch + 1
    if mismatch ≤ k
      then print i
  ```

- **Runtime ?**

Dynamic Programming

- **Algorithm**
  - Count mismatches, \( D(i,j) \), between \( P[i] \) and \( T[j] \)
  - Find all positions \( i \) such that the number of mismatches between \( P[1..m] \) and \( T[l..(l+m-1)] \) is less than or equals to \( k \)

  \[
  D(i,j) = \begin{cases} 
  D(i-1,j-1) + 0 & \text{if } P[i] = T[j] \\
  D(i-1,j-1) + 1 & \text{otherwise}
  \end{cases}
  \]

- **Example**
  - \( T = "AGCCTTGA" \), \( P = "GCAT" \), \( k=2 \)

- **Runtime ?**
Overview

- Pattern Matching
  - Exhaustive Search
  - FA Algorithm
  - KMP Algorithm

- Multiple Pattern Matching
  - AC Algorithm
  - Weiner’s Algorithm

- Approximate Pattern Matching

- Pattern Finding

- Motif Finding (Approximate Pattern Finding)

Pattern Matching vs. Pattern Finding

- Pattern Matching
  - Given a set of known patterns (motifs),
  - Find all occurrences in a DNA or protein sequence
  - Determine function of the DNA or protein sequence

  $\text{agctccgatgtaagtcacatgtaatctcgatgtaacctaccttacctacagtacataca}$

  $p = \text{"cgatgt"}$

  $\text{agctccgatgtaagtcacatgtaatctcgatgtaacctaccttacctacagtacataca}$

- Pattern (Motif) Finding
  - Given a set of known DNA or protein sequences,
  - Find patterns (motifs) across the DNA or protein sequences
Example of Pattern Finding

- Example
  - "acgtacgt" occurs in all 5 sequences (100%)

- Challenges
  - We do not know how the pattern looks like ("random" patterns)
  - We do not know where the pattern is located in each gene sequence

Formulation of Pattern Finding Problem

- Goal
  - Finding all occurrences of substrings (length-$m$) that appear in all strings (length-$n$)

- Input
  - A set of $k$ strings $T_1, T_2, \ldots, T_k$ and the length of the substring, $m$

- Output
  - The set of starting position vectors, $s = (s_1, s_2, \ldots, s_k)$, of the substrings that appear in all $k$ strings
Naïve Approach

Algorithm
- Exhaustive search among all possible \( s \)

\[
\text{NAIVEFINDING}(\{T_1, T_2, \ldots, T_k\}, n, m) \\
\text{for } s = (1, 1, \ldots, 1) \text{ to } (n - m + 1, \ldots, n - m + 1) \\
\text{if } T_1[s_1, \ldots, s_1 + m - 1] = T_2[s_2, \ldots, s_2 + m - 1] = \cdots = T_k[s_k, \ldots, s_k + m - 1] \\
\text{then print } s
\]

Runtime ?

Alternative Naïve Approach

Algorithm
- Exhaustive search among all possible combinations of characters
- Practical in biological applications

Search Tree
- Example where \( m=2 \) in DNA sequences

Runtime?
Anti-Monotonic Approach

- Anti-Monotonic Property
  - If a set (a string) violates a rule or a constraint (a pattern), then all super-sets (super-strings) violate it.

- Process of Pattern Finding
  - Finding the patterns (substrings) by increment of their length.
  - If a sub-pattern does not appear in all strings $T_i$ where $1 \leq i \leq k$, do not process further its super-patterns.

Overview

- Pattern Matching
  - Exhaustive Search
  - FA Algorithm
  - KMP Algorithm

- Multiple Pattern Matching
  - AC Algorithm
  - Weiner’s Algorithm

- Approximate Pattern Matching

- Pattern Finding

- **Motif Finding (Approximate Pattern Finding)**
Approximate Pattern Finding

- **Example**
  
  
  \[
  \begin{align*}
  T_1 &= \text{ctgatagncgtatcgctatccacgtacgtaggtctctgtgcgaatctabgcgttcaccccat} \\
  T_2 &= \text{agtactggtgtacatbtgatacgtacgphacaccggcaacctgaanccacgtcagaaccaccaggtgc} \\
  T_3 &= \text{aaacgtacgtgcaccctcttttctttctgctgctgcacaggcaggtgtatagacgaaaat} \\
  T_4 &= \text{agccctccgatgtagcatagctgtaactattacctgccacccctattacatcttacgcacgtataca} \\
  T_5 &= \text{ctgttatacaacgctactgggtatgctgttttgctgtagctcgatacgtacgtc}
  \end{align*}
  \]

  - "acgtacgt" occurs in no sequences (0%) after mutations

- **Challenges**
  
  - We do not know how the motif looks like ("random" motifs)
  - We do not know where the motif is located in each gene sequence
  - Motifs can differ slightly from one gene to another by mutations

Consensus Pattern & Consensus Score

- **Profiles**
  
  - Frequency of each character on each column of \( s \)
  
  \[
  \begin{align*}
  \text{Alignment} & \quad \text{Profile} \\
  aGgtatcTt & \quad A \ 3 \ 0 \ 1 \ 0 \ 3 \ 1 \ 1 \ 0 \\
  CcAatatcgt & \quad C \ 2 \ 4 \ 0 \ 0 \ 1 \ 4 \ 0 \ 0 \\
  acgtTAgt & \quad G \ 0 \ 1 \ 4 \ 0 \ 0 \ 0 \ 3 \ 1 \\
  acgtCCgt & \quad T \ 0 \ 0 \ 0 \ 5 \ 1 \ 0 \ 1 \ 4 \\
  cgcatacgG & \quad \rightarrow \text{Consensus motif}
  \end{align*}
  \]

- **Consensus Pattern**
  
  - A string such that each column has the character with the highest frequency

- **Consensus Score**
  
  - \( C(i) \): the largest count in column \( i \)
  
  \[
  C_r(s) = \sum_{i=1}^{m} C(s_i)
  \]
Formulation of Motif Finding Problem (1)

➢ Goal
  ▪ Finding approximate occurrences of substrings (length-$m$) from
    the set of strings (length-$n$)

➢ Input
  ▪ A set of $k$ strings $T_1, T_2, \ldots, T_k$ and the length of the substring, $m$

➢ Output
  ▪ The starting position vector, $s = (s_1, s_2, \ldots, s_k)$, of the substrings that
    maximize the consensus score over all $k$ strings

Naïve Approach

➢ Algorithm
  ▪ Exhaustive search

  ```
  NAIVEMOTIFFINDING([T_1, T_2, \ldots, T_k], n, m)
  score ← 0
  pos ← (0, 0, \ldots, 0)
  for $s = (1, 1, \ldots, 1)$ to $(n - m + 1, \ldots, n - m + 1)$
    if $C_T(s)$ > score
      then score ← $C_T(s)$ and pos ← $s$
  return pos
  ```

➢ Runtime ?
Total Distance

- Definition
  - \( d_H(u, v) \): hamming distance between two strings, \( u \) and \( v \)
  - Given a pattern \( P \), total distance, \( d_T(P, s) = \sum_{i=1}^{k} d_H(P, T[s_i, \ldots, (s_i + m - 1)]) \)

- Example
  - \( P = \) "ATGCAACT", \( s = (s_1, s_2, s_3) \)
  - \( T_1[s_1,\ldots,(s_1+7)] = \) "ATCCAGCT"
  - \( T_2[s_2,\ldots,(s_2+7)] = \) "AAGCAACC"
  - \( T_3[s_3,\ldots,(s_3+7)] = \) "ATGCCATT"
  - \( d(P, s) = ? \)

- Relationship between Total Distance and Consensus Score?

Formulation of Motif Finding Problem (2)

- Goal
  - Finding approximate occurrences of substrings (length-\( m \)) from the set of strings (length-\( n \))

- Input
  - A set of \( k \) strings \( T_1, T_2, \ldots, T_k \) and the length of the substring, \( m \)

- Output
  - The starting position vector, \( s = (s_1, s_2, \ldots, s_k) \), of the substrings that minimize the total distance over all \( k \) strings
Naïve Approach

- Algorithm
  - Exhaustive search

```python
NAIVE-MOTIF-FINDING(\{T_1, T_2, \ldots, T_k\}, n, m)
  dist \leftarrow \infty
  \text{for each pattern } P
    \text{for each } i
      \text{find } \min_{\hat{d}_H(P, T[i, \ldots, i + m - 1])}
      s_i \leftarrow x
      \text{if } d_T(P, s) \text{ < dist}
        \text{then } dist \leftarrow d_T(P, s) \text{ and } pos \leftarrow s
  \text{return } pos
```

- Runtime?

Branch-And-Bound Approach

- Process
  - Run the naïve motif finding algorithm with \( m' \) where \( m' < m \)
  - Output all starting position vectors \( s \) where \( d_T(P, s) < \) max threshold
  - Run the naïve motif finding algorithm with \( m \) for \( s \)

- Problem?
  - Trade-off in efficiency vs. accuracy
Questions?

- Lecture Slides are found on the Course Website,
  web.ecs.baylor.edu/faculty/cho/5330