Today: Exact String Matching

1. Introduction
   - The exact string matching problem
   - Naïve algorithm
2. Making bigger strides
   - Fundamental pre-processing
   - Knuth-Morris-Pratt / Boyer-Moore / Z-algorithm
3. Suffix-trees
   - Extreme pre-processing
   - Linear time construction and applications
4. Semi-numerical string matching
   - Rabin-Karp algorithm
   - Leads to hashing and the Blast algorithm

Properties of high-scoring alignments

- Few gaps
  - Strong matches should stay close to the diagonal
  - Can search only regions close to the diagonal
  ➔ Bounded dynamic programming
- Few mismatches
  - There should be stretches of perfect identity
  - Can search for alignments containing such segments
  ➔ Hash k-mers: BLAST

Finding a query sequence in a large database

- We could just scan the whole database
  - Simply align the query to every sequence in turn
  - Report highest-scoring sequences
- But:
  - Query must be very fast
  - Most sequences will be completely unrelated to query
  - Individual alignment needs not be perfect. Can fine-tune
  - Exploit nature of the problem
    - If you’re going to reject any match with idperc < 90%, then why bother even looking at sequences which don’t have a fairly long stretch of matching a.a. in a row.

The exact matching problem

- Inputs:
  - a string \( P \), called the pattern
  - a longer string \( T \), called the text
- Output:
  - Find all occurrences, if any, of pattern \( P \) in text \( T \)
- Example

\[
P = a b a
\]

\[
T = \begin{array}{cccccccccc}
  & a & b & a & c & a & b & a & a & d \\
 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12
\end{array}
\]
Basic string definitions

- A string $S$  
  - Ordered list of characters  
  - Written contiguously from left to write

- A substring $S[i..j]$  
  - all contiguous characters from $i$ to $j$  
  - Example: $S[3..7] = abaxa$

- A prefix is a substring starting at 1
- A suffix is a substring ending at $|S|$
- $|S|$ denotes the number of characters in string $S$

The naïve string-matching algorithm

- NAÏVE STRING MATCHING
- $n \leftarrow$ length[$T$]  
- $m \leftarrow$ length[$P$]  
- for shift $\leftarrow 0$ to $n$
  - do if $P[1..m] \equiv T[shift+1 .. shift+m]$
  - then print “Pattern occurs with shift” shift
- Where the test operation in line 4:
  - Tests each position in turn  
  - If match, continue testing  
  - else: stop
- Running time $\sim$ number of comparisons
  - number of shifts (with one comparison each)  
  + number of successful character comparisons

Computations made with naïve algorithm

- Worst case running time:  
  - Test every position  
  - P=aaaa, T=aaaaaaaaaa
- Best case running time:  
  - Test only first position  
  - P=bbbb, T=aaaaaaaaa

Key insight: make bigger shifts!

- If all characters in the pattern are the same:
  - Information gathered at every comparison  
  - Knowledge of the internal structure of $P$
  - Number of comparisons: $O(n)$

Key insight: make bigger shifts!

- If all characters in the pattern are different:
  - At most $n$ matching comparisons
  - At most $n$ non-matching comparisons
  - Number of comparisons: $O(n)$

Key insight: make bigger shifts!

- Special case:
  - If all characters in the pattern are the same: $O(n)$
  - If all characters in the pattern are different: $O(n)$
- General case:
  - Learn internal redundancy structure of the pattern
  - Pattern pre-processing step
- Methods:
  - Fundamental pre-processing
  - Knuth-Morris-Pratt
  - Finite State Machine
2. String Pre-processing

Fundamental pre-processing

• Learning the redundancy structure of a string $S$

Let $S$ be a string and $Z$ be its transformation.

- $Z_i = \text{length of longest prefix in common for } S[i..] \text{ and } S$

The length of the longest prefix of $S[i..]$ that's also a prefix of $S$.

- $S = aabc aabx$
- $Z = \text{Z-box} = aabc aabx$
- $r = aabc aabx$
- $l = aabc aabx$
- $k_{\text{left}} = aabc aabx$
- $k_{\text{right}} = aabc aabx$

Can we compute $Z, r, l$ in linear time $O(|S|)$?

Computing $Z_k$ given $Z_1 .. Z_{k-1}$

- Case 1: $k$ is outside a Z-box: simply compute $Z_k$

- Case 2: $k$ is inside a Z-box: Look up $Z_k$

- Case 2a: $Z_k' < r-k$
- Case 2b: $Z_k' \geq r-k$

Putting it all together

- FUNDAMENTAL-PREPROCESSING($S$):
  - $Z_2, l, r = \text{explicitly compare } S[1..] \text{ with } S[2..]$
  - for $k$ in 2..n:
    - if $k > r$: $Z_k, l, r = \text{explicitly compare } S[1..] \text{ with } S[k..]$
    - if $k < r$:
      - if $Z_k' < (r-k)$:
        - $Z_k = Z_k'$
      - else:
        - $Z_k = \text{explicitly compare } S[r+1..] \text{ with } S[(r-k)+1..]$
      - $l = k$
      - $r = l + Z_k$

Explicitly compare starting at $r+1$
Correctness of Z computation
Case 1: k is outside a Z-box: explicitly compute Z_k
Case 2a: Inside Z-box and Z_k < r-k: set Z_k = Z_k'
Case 2b: Inside Z-box and Z_k' >= r-k: explicitly compute starting at r+1

What’s so fundamental about Z?
- Learning the redundancy structure of a string S
- Z_i = fundamental property of internal redundancy structure
- Most pre-processings can be expressed in terms of Z
  - Length of the longest prefix starting/ending at position i
  - Length of the longest suffix starting/ending at position i

Back to string matching
- Given the fundamental pre-processing of pattern P
  - Compare pattern P to text T
  - Shift P by larger intervals based on values of Z
- Three algorithms based on these ideas
  - Knuth-Morris-Pratt algorithm
  - Boyer-Moore algorithm
  - Z algorithm

Knuth-Morris-Pratt algorithm
- Pre-processing:
  - S_P(P) = length of longest proper suffix of P[1..i] that matches a prefix of P
  - No other than the right-hand-side of the Z-boxes

Knuth-Morris-Pratt running time
- Number of comparisons bounded by characters in T
  - Every comparison starts at text position where last comparison ended
  - Every shift results in at most one extra comparison
  - At most |T| shifts \( \Rightarrow \) Running time bounded by \( 2|T| \)
Boyer-Moore algorithm

Three fundamental ideas:
1. Right-to-left comparison
2. Alphabet-based shift rule
3. Preprocessing-based shift rule

Results in:
- Very good algorithm in practice
- Rule 2 results in large shifts and sub-linear time
  - for larger alphabets, e.g., English text
- Rule 3 ensures worst-case linear behavior
  - even in small alphabets, e.g., DNA sequences

The Z algorithm

The Z algorithm
- Concatenate P + '$' + T
- Compute fundamental pre-processing O(m+n)
- Report all starting positions i for which Z_i = |P|

What have we learned so far?

Making bigger strides
- Fundamental preprocessing in linear time
- Searching for pattern p in linear time: O(Text)

Today's challenge: Can we do better?
- Searching for any pattern p in linear time O(pattern)
- After pre-processing the text once, in linear time!

3. Suffix Trees

More involved pre-processing step

- Fundamental pre-processing only searched for:
  - Common prefix / suffix at any position
  - Redundancy with beginning/end of string
- Suffix trees
  - Redundancy across all substrings
    - starting at every position
    - over the remainder of the list
- Example:
  - Suffix tree of xabxac

Suffix tree definition

- Definition: Suffix tree T for string S (of length n)
  - Rooted, directed tree T, n leaves, numbered 1..n
  - Path to leaf i spells out the suffix S[i..], by concatenating edge labels
  - Common prefixes share common paths, diverge to form internal nodes
  - Effectively exhibit common prefixes of every suffix
  - Explores full substring redundancy structure of S
Exact string matching with suffix trees

- Given the suffix tree for text T
- Search pattern P in O(pattern) time
  - For every character in P, traverse the appropriate path of the tree, reading one character each time
  - If P is not found in a path, P does not occur in T
  - If P is found in its entirety, then all occurrences of P in T are exactly the children of that node
    - Every child corresponds to exactly one occurrence
    - Simply list each of the leaf indices

Suffix Tree Construction

- Naïve algorithm: Insert one suffix at a time
  - For i in [1..n]:
    - # insert suffix S[i:end]
    - for each character in S[i..n]:
      - case 1: (end of edge label)
        - extend the edge label
      - case 2: (differing character)
        - insert new branch
      - case 3: (char already exists)
        - do nothing
  - Running time: O(n²)

High-level description

- UKKONEN-ALGORITHM
  - Construct I₁
  - for i in [1..n-1]:
    - (begin phase i+1)
      - for j in [1..i+1]:
        - (begin extension j+1)
          - Find the end of the path labeled S[j..i] in current tree (call it β)
          - if [case 1]: β ends in a leaf
            - then add S[i+1] to that edge label
          - if [case 2]: β ends in middle of edge
            - then add an internal node and new branch
          - if [case 3]: S[i+1] already exists
            - then do nothing
          - ensure we only back up by one node at each iteration
  - Running time: O(n³)

Can we do better?

Suffix Tree Construction

The three cases for suffix tree extension

- Case #1: reached end of existing edge label:
  - add a character to the end of an edge label
- Case #2: reached mismatch in the middle of a branch
  - split the branch: add internal node and branch leaving it
- Case #3: entire substring already exists
  - do nothing

Speedup the search for β: Use suffix links

- Ukkonen’s algorithm (with suffix links)
- Running time: O(n²)?
  - Not yet, but almost there!
Speedup the search for $\beta$: Use suffix links

- Link between internal nodes. From $\text{c'}+\alpha \to \alpha$
  
  - How does this help?
    - We ensure we only backtrack by 1 node, before traversing $(v,s(v))$
    - Node-depth decreases by another 1 node, during traversal
      - since $\text{c'}u$ is a substring of $\alpha$, all branches are identical
    - Thus, over an entire phase, at most $O(n)$ increases
      - max node-depth is $O(n)$, bounded by the number of characters
    - at most $2n$ decrements
      - at most $3n$ increments

  - Must ensure constant traversal time per node!

Summary: Linear time for each phase

- $O(n)$ node traversals over the entire phase
  - No more than $2n$ decreases in node-depth
  - By using $v,s(v)$ links, avoid backtracking to root
  - Longest depth is $n$ (length of string)
    
- $O(n)$ node traversals per phase
  - Constant time per node traversal
    - Fixed alphabet
      - Indexing of branches to follow at every node
    - Known length to traverse for every edge
      - no need for explicit character comparisons
  - $O(n)$ time over the entire phase
  - $O(n^2)$ time over the entire construction

Can we do better?

Three more ideas

(a) Edge representation
  - Use (start,end) indexing to $S$  

(b) Once a skip, always a skip: case #3 terminates entire phase
  - If prefix is already included, entire extension already included
    - When case #3 applies once, it applies for entire phase

(c) Once a leaf, always a leaf: case #1 work done centrally
  - Special symbol for end of $S$  

b. Reducing the work for case #3

- Case #1: add a character to the end of an edge label
- Case #2: add a new branch and internal node
- Case #3: do nothing (path already exists)

If case #3 applies to extension $j$, then it applies to all $j' > j$. Since the entire suffix $S[j..i]$ is already included in the tree
  
  - case #3 terminates an entire phase

Example:

<table>
<thead>
<tr>
<th>Phase</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b c d</td>
<td>e</td>
<td>f</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>u</td>
<td>v</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Constant space for every node. $O(n)$ space for entire tree
c. Skipping the work for case #1

- Extension case #1: add a character to the end of an edge label

- Once a leaf always a leaf:
  - If case #1 applies for character $j$ at some phase, it applies for $j$ in all subsequent phases
  - Initial stretch of case #1 applications at beginning of each phase
  - Centralize this work
  - Create “current end” variable $e$, updated centrally at start of each phase, extending all leaves
  - Start explicit work after this initial stretch (at $j^*$ where case #3 first applied in previous phase)

- Implicit work skipped (after case #3)

Amortized constant cost per phase!

- Summary: three smart ideas
  - (a) Constant space per edge (start,end) variables
  - Centralized end-of-string variable $e$
  - (b) Once a skip, always a skip
  - Case #3 terminates phase, since substring already included
  - (c) Once a leaf, always a leaf
  - Case #1 centralized, for entire start of each phase
  - Constant amortized cost per phase
  - Linear cost for entire algorithm!

Putting it all together

- UKKONEN-ALGORITHM
  - Construct $I_i$
  - for $i$ in [1..m-1]:
    - (begin phase $i+1$)
      - Increment end variable $e$ [centralized case #1]
    - (begin explicit work where last case #3 was applied $j^*$)
      - Use link $v \rightarrow s(v)$ to find $S[j..i]$ in current tree (call it $\beta$) [suffix-links]
      - if [case1]: $\beta$ ends on a leaf
        - do nothing [case #1 is now centralized]
      - if [case2]: $\beta$ ends in middle of edge
        - add an internal node and new branch
      - if [case3]: $i(i+1)$ already exists
        - then end entire phase, set $f$

Applications of suffix trees

- Exact string matching
- Dictionary lookup for one word
- Search of multiple strings for one pattern
- Longest common substring problem
- Common substrings of more than two strings
- Longest common extension
- Inexact string matching

4. Semi-numerical string matching

Karp-Rabin algorithm

- Key idea:
  - Interpret strings as numbers: fast comparison
Karp-Rabin algorithm

- **Key idea:**
  - Interpret strings as numbers: fast comparison
- **To make it work:**
  - Compute next number based on previous one $O(1)$
  - Hashing (mod $p$) $O(1)$

\[
x = 31,415
\]

\[
y_1 = 23,590
g_2 = 35,902
\]

\[
T = 2 3 5 9 0 2 3 1 4 1 5 2 6 7 3 9 9 2 1
\]

$\mod p$ (ex: $p=13$)

\[
\begin{array}{cccccccccccccc}
9 & 9 & 9 & 7 & 8 & 1 & 1 & 1 & 7 & 4 & 0 & 1 \\
\end{array}
\]

- **Consequences of (mod $p$) ‘hashing’**
  - Good: Enable fast computation (use small numbers)
  - Bad: Leads to spurious hits (collisions)

\[
	ext{(this actually works)}
\]

- Complete algorithm must deal with the bad

Karp Rabin key idea: Semi-numerical approach

- **Idea 1: semi-numerical approach:**
  - Consider all m-mers: $T[1...m], T[2...m+1], ..., T[m+n-1...n]$
  - Map each $T[s+1...s+m]$ into a number $t_s$
  - Map the pattern $P[1...m]$ into a number $p$
  - Report the m-mers that map to the same value as $p$

Semi-numerical approach: implementation

- **First attempt:**
  - Assume $\Sigma=\{0,1\}$
  - Think about each $T[s+1...s+m]$ as a number in binary representation, i.e.,
    \[
    t_s = T[s+1]2^{m-1} + T[s+2]2^{m-2} + ... + T[s+m]2^0
    \]
  - Output all $s$ such that $t_s$ is equal to the number $p$ represented by $P$

- **Problem: how to map all m-mers in $O(n)$ time?**
  - Find a fast way of computing $t_s$, given $t_n$

Idea 2: Computing all numbers in linear time

- **How to transform**

\[
t_s = T[s+1]2^{m-1} + T[s+2]2^{m-2} + ... + T[s+m]2^0
\]

Into

\[
t_{s+1} = T[s+2]2^{m-1} + T[s+3]2^{m-2} + ... + T[s+m+1]2^0
\]

- **Can compute $t_{s+1}$ from $t_s$, using 3 arithmetic operations:**
  - Subtract $T[s+1]2^{m-1}$
  - Multiply by 2 (i.e., shift the bits by one position)
  - Add $T[s+m+1]2^0$

- **Therefore:**

\[
t_{s+1} = (t_s - T[s+1]2^{m-1})2 + T[s+m+1]2^0
\]

- **Therefore, we can compute all $t_{s+1}, t_{s+2}, ..., t_n$ using $O(n)$ arithmetic operations, and a number for $P$ in $O(m)$**

Problem: Long strings = big numbers

- To get $O(n)$ time, we would need to perform each arithmetic operation in $O(1)$ time
- However, the arguments are m-bit long!
- If $m$ large, it is unreasonable to assume that operations on such big numbers can be done in $O(1)$ time
- We need to reduce the number range to something more manageable
Idea 3: Hashing

- We will instead compute:
  \[ t'_s = T[s+1]2^{m-1} + T[s+2]2^{m-2} + \ldots + T[s+m]2^0 \pmod q \]
  where q is an "appropriate" prime number

- One can still compute \( t'_{s+1} \) from \( t'_s \):
  \[ t'_{s+1} = (t'_s - T[s+1]2^{m-1})2 + T[s+m+1]2^0 \pmod q \]

- If q is not large, we can compute all \( t'_s \) (and \( p' \)) in \( O(n) \) time

Problem: hashing leads to false positives

- Unfortunately, we can have false positives, i.e.,
  \( T[s+1...s+m] \neq P \) but \( t'_s \pmod q = p \pmod q \)

- Our approach:
  - Use a random q
  - Show that the probability of a false positive is small
    → randomized algorithm

Karp-Rabin algorithm: Putting it all together

1. Semi-numerical: interpret strings as numbers
2. Use previous score to compute the next one
3. Hashing keeps numbers small, computation fast

*Example:
\[
\begin{align*}
3 &\quad 1 &\quad 4 &\quad 1 &\quad 5 &\quad 2 \\
&\quad &\quad &\quad &\quad &\quad \\
14152 &\equiv (31415-3\cdot10000) \cdot 10 + 2 \pmod{13} \\
&\equiv (7-3\cdot3) \cdot 10 + 2 \pmod{13} \\
&\equiv 8 \pmod{13}
\end{align*}
\]

- Other semi-numerical methods
  - Fast Fourier Transform
  - Shift-And method

Today: Exact String Matching

1. Introduction
   - The exact string matching problem
   - Naive algorithm

2. Making bigger strides
   - Fundamental pre-processing
   - Knuth-Morris-Pratt / Boyer-Moore / Z-algorithm

3. Suffix-trees
   - Extreme pre-processing
   - Linear time construction and applications

4. Semi-numerical string matching
   - Rabin-Karp algorithm
   - Leads to hashing and the Blast algorithm