Seminumerical String Matching

CMSC 701

Something completely different…

Semi-numerical string matching:

Instead of focusing on comparing characters, think of string as a **sequence of bits or numbers** and use arithmetic operations to search for patterns.

Two algorithms:

- Rabin-Karp
- Shift-And

Both tend to be better for short patterns.

Rabin-Karp

(Following CLR Chapter 34)

Characters as digits

- Assume $\Sigma = \{0,\ldots,9\}$
- Then a string can be thought of as the decimal representation of a number:

427328

- In general, if $|\Sigma| = d$, a string represents a number in base d.
- Let p = the number represented by query P.
- Let t_s = the number represented by the $|P|$ digits of T that start at position s.

P occurs at position s of $T \Leftrightarrow p = t_s$.

Computing p and t_s

- Use Horner's rule to compute p in time $O(|P|=m)$:

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + \dots + 10(P[2] + 10P[1])\dots)$$

- Example: $427328 = (8+10(2+10(3+\underset{\substack{\uparrow \\ \text{“Left shift” by 1 digit}}}{10}(7+10(2 + 10 \times 4)))))$

- t_0 can be computed the same way in time $O(|P|=m)$.

- t_s can be computed from t_{s-1} in $O(1)$ time:

$$t_s = \underset{\text{shift left by 1 digit}}{10}\underbrace{(t_{s-1} - 10^{m-1}T[s-1])}_{\text{remove high-order digit}} + \underbrace{T[s+m-1]}_{\text{add next digit of T as the low-order digit}}$$

Rabin-Karp

Compute p.

Iteratively compute t_s.

Output s when $t_s = p$.

Problem: p and t_s might be huge numbers.

Solution: compute everything modulo some prime q.

- If $10q$ is $\leq$ word size, then p mod q and t_s mod q can be computed in a single word.
- If p occurs at t_s, then $p \equiv t_s \pmod{q}$

New problem: If $p \equiv t_s \pmod{q}$, it doesn't necessarily mean there is a match at s.

New solution: if $p \equiv t_s \pmod{q}$, check match explicitly.

Worst-case runtime = $O(mn)$, if every position is a match or false positive.

Rabin-Karp Notes

- If your pattern is very small, don't need to use the (mod q) trick, and you can avoid false positive matches.
- You can also pick several different primes q_1, q_2,..., q_k and then require that:

$$p \equiv t_s \pmod{q_1}$$

$$p \equiv t_s \pmod{q_2}$$

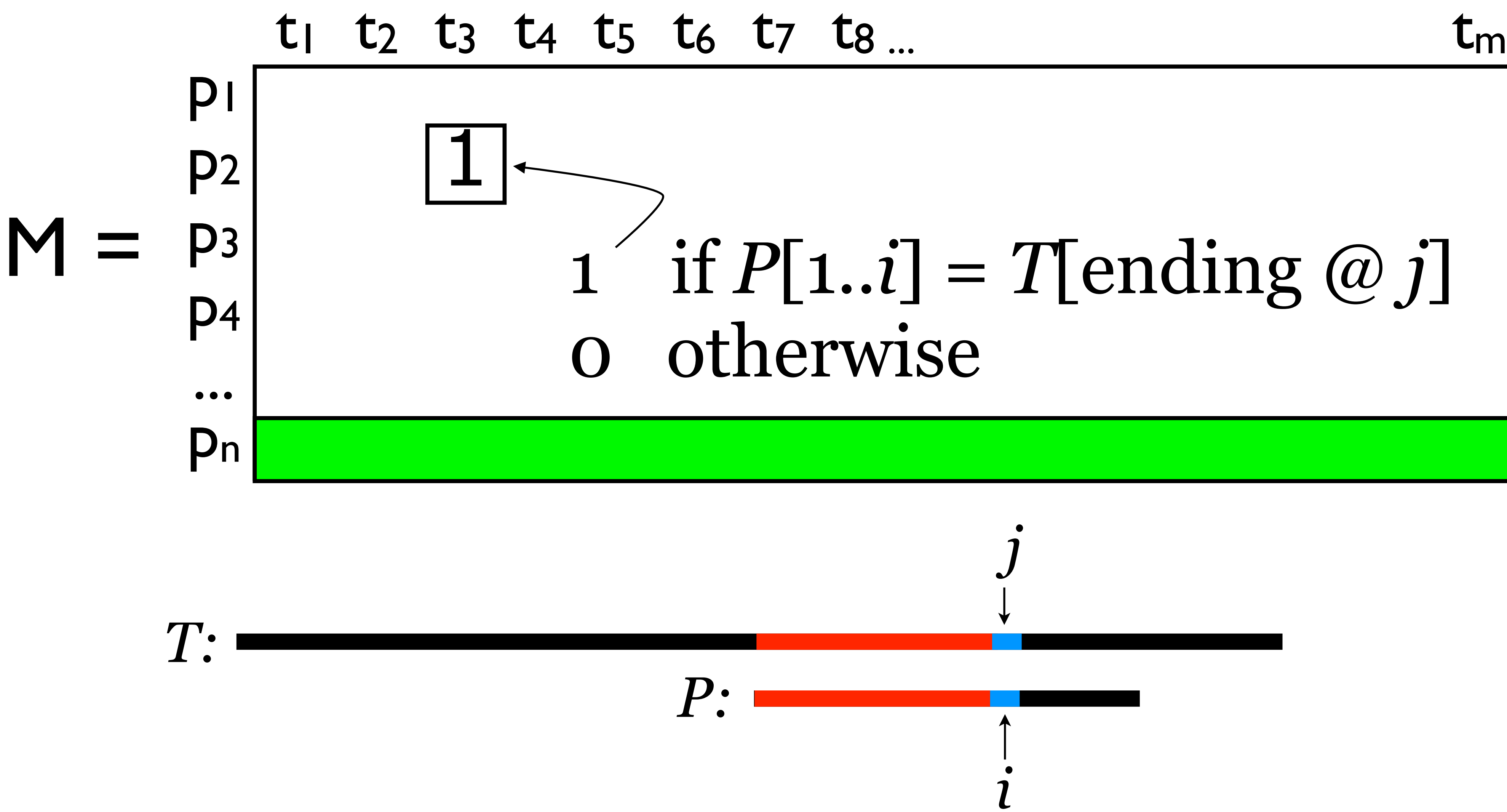
$$\vdots$$

$$p \equiv t_s \pmod{q_k}$$

Shift-And
(Following Gusfield Chapter 4)

Shift-And Algorithm

$M[i,j]$:= 1 iff prefix i of P matches a substring of T ending at j:



1's in last row will indicate where P matches T.

Decompose computation of $M[i,j]$:

$$M[i,j] = (P[i] = T[j]) \textbf{ and } \underbrace{(P[1..i\text{-}1] = T[\text{ending @ } j\text{-}1])}_{M[i\text{-}1,\, j\text{-}1]}$$

Computing M by columns

$$M[i,j] = P[i] = T[j] \textbf{ and } \underbrace{P[1..i\text{-}1] = T[\text{ending @ } j\text{-}1]}_{M[i\text{-}1,\, j\text{-}1]}$$

Def. $U_P(x)$ = $|P|$-bit vector where i^{th} entry is 1 if $P[i] = x$, 0 otherwise.

Compute columns of M left to right:

$$M[\bullet,j] = U_P(T[j]) \;\&\; (1;\, M[\bullet,j\text{-}1])$$

j^{th} column of M

1 where $P[i] = T[j]$

previous column of M, shifted down by 1 (prepended with a 1)

$$\begin{bmatrix}0\\0\\0\\1\end{bmatrix} = \begin{bmatrix}0\\0\\1\\1\end{bmatrix} \;\&\; \begin{bmatrix}1\\1\\0\\1\end{bmatrix}$$

first entry always 1 because red condition is empty for $i = 1$.

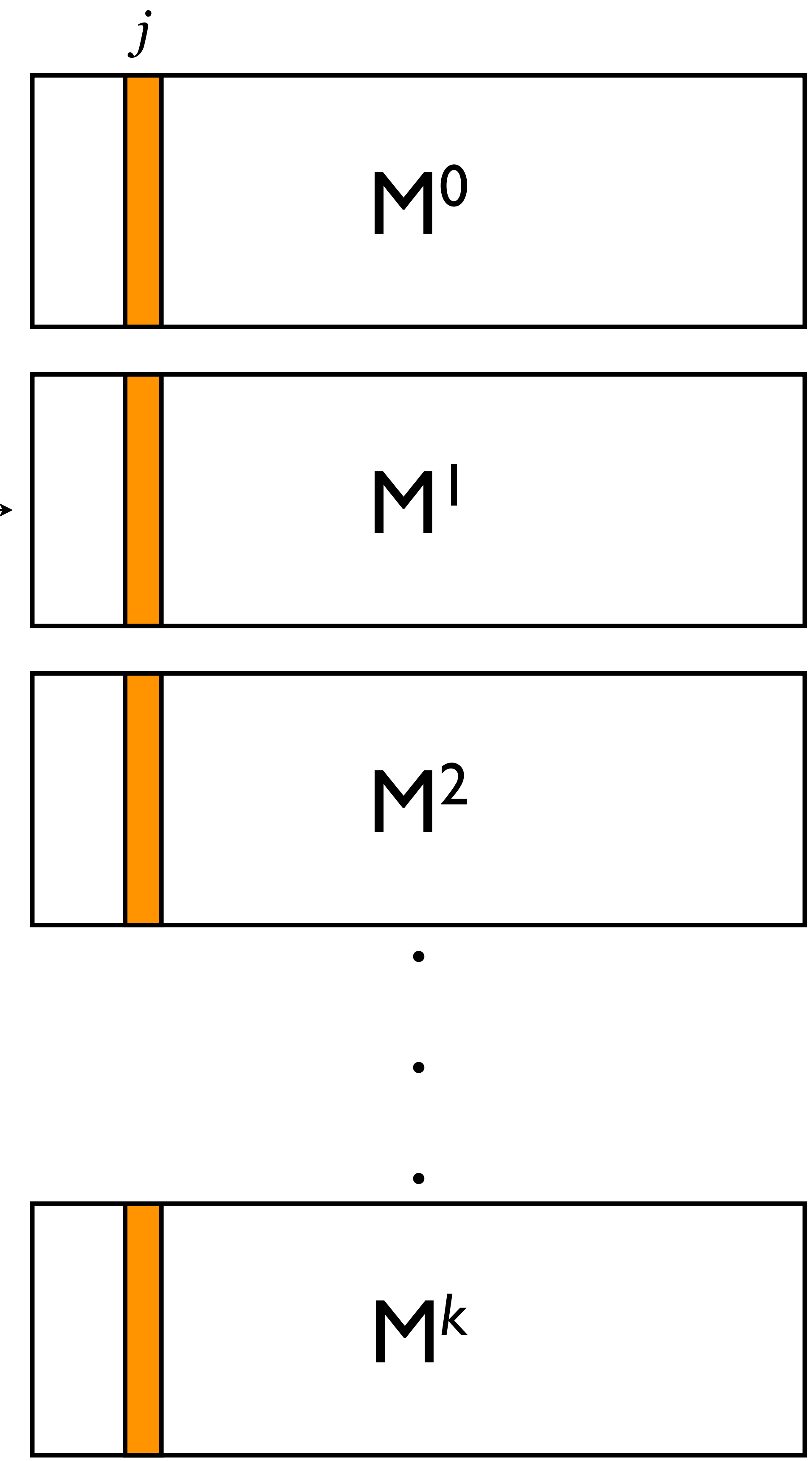
Shift-And Time & Space

- Only the current and previous columns of M are needed, so space is O($|P|$).
- Worst case running time O($|P| \times |T|$).
- But if $|P|$ in bits ≤ computer word, each column of M can be computed in constant time, leading to an O($|T|$) algorithm.

Extension to approximate matching

bs(v) := (1; v) truncated to n dimensions.

$M^l[i,j]$ = i^{th} prefix of P matches suffix ending at j of T **with ≤ l mismatches.**



i^{th} prefix of P matches string ending at j with ≤ l-1 mismatches

i^{th} prefix of P matches string ending at j-1 with ≤ l-1 mismatches

$$M^l[j] = M^{l-1}[j] \textbf{ or } \underbrace{\Big(\mathrm{bs}(M^l(j-1)) \textbf{ and } U(T(j))\Big)}_{\text{i-1 characters of P match with $\leq l$ mismatches and j^{th} character matches.}} \textbf{ or } M^{l-1}[j-1]$$

Seminumerical Matching

Often effective when pattern is small.

Asymptotically, not the best run time, but if operations can be done fast in hardware, these algorithms can be good choices.

Also, provide a different perspective on the string matching problem.