String algorithms

"Stringology"

Exact string matching

Pattern search in a sequence

- T[1..n] sequence (text, string)
- P[1..m] pattern
- *Problem*: locate all occurrences of P in T (*variants*: check if P occurs in T, count the number of occurrences)

Naïve algorithm: O($n \cdot m$)

T=aaaaaa...aa=a^n

P=aa...ab=a^{m-1}b

Pattern search in a sequence

```
T = a b a b a a a b a b a a b a …
P = a b a a b
```

Pattern search in a sequence

T = a b a b a a a b a b a a b a …

P = a b a a b

Pattern search in a sequence

T = a b a b a a a b a b a a b a …

P = a b a a b

Pattern search in a sequence

T = a b a b a a a b a b a a b a …

P = a b a a b

Pattern search in a sequence

T = a b a b a a a b a b a a b a …

P = a b a a b

Pattern search in a sequence

```
T = a b a b a a a b a b a a b a …
P = a b a a b
        a b a a b
```

Pattern search in a sequence

T = a b a b a a a b a b a a b a …

P = a b a a b

a b a a b

Pattern search in a sequence

T = a b a b a a a b a b a a b a …

P = a b a a b

a b a a b

Pattern search in a sequence

```
T = a b a b a a a b a b a a b a …
P = a b a a b
        a b a a b
```

Pattern search in a sequence

T = a b a b a a a b a b a a b a …

P = a b a a b

a b a a b

a b a a b

Pattern search in a sequence

```
T = a b a b a a a b a b a a b a ...
P = a b a a b
        a b a a b
              a b a a b
```

Pattern search in a sequence

T = a b a b a a a b a b a a b a …

P = a b a a b

a b a a b

a b a a b

Pattern search in a sequence

T = a b a b a a a b a b a a b a …

P = a b a a b

a b a a b

a b a a b

Pattern search in a sequence

```
T = a b a b a a a b a b a a b a …
P = a b a a b
        a b a a b
                a b a a b
                    a b a a b
```

Pattern search in a sequence

```
T = a b a b a a a b a b a a b a …
P = a b a a b
        a b a a b
                a b a a b
                    a b a a b
```

Pattern search in a sequence

T = a b a b a a a b a b a a b a …

P = a b a a b

a b a a b

a b a a b

a b a a b

Pattern search in a sequence

```
T = a b a b a a a b a b a a b a ...
P = a b a a b
        a b a a b
                a b a a b
                    a b a a b
```

Pattern search in a sequence

```
T = a b a b a a a b a b a a b a …
P = a b a a b
        a b a a b
                a b a a b
                    a b a a b
                          a b a a b
```

Pattern search in a sequence

```
T = a b a b a a a b a b a a b a …
P = a b a a b
        a b a a b
                a b a a b
                    a b a a b
                          a b a a b
```

Pattern search in a sequence

Observations:

• at each step we move on by one letter in the text (i.e. each letter of the text is compared once)
• the state of the search is defined by a position in the text and a position in the pattern
• the shift of the pattern is defined depending on the failure position in the pattern and the text letter produced the failure

```
T = … * * * * * * * * …
        = = = ≠
P =     * * * * *
```

Pattern search in a sequence

Observations:

• at each step we move on by one letter in the text (i.e. each letter of the text is compared once)
• the state of the search is defined by a position in the text and a position in the pattern
• the shift of the pattern is defined depending on the failure position in the pattern and the text letter produced the failure

```
T = … * * * * * * * * …
        = = = ≠
P =     * * * * *
```

$\Rightarrow$ Finite automaton!

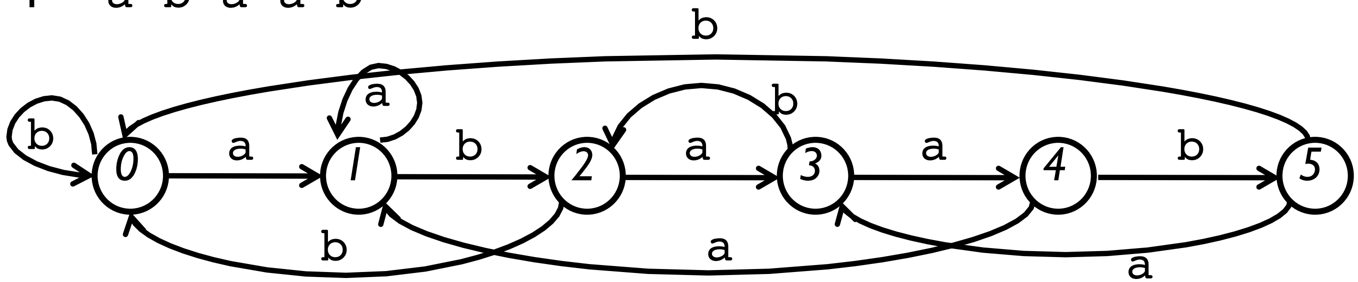
Pattern search in a sequence

P $\Rightarrow$ finite automaton

state ≡ prefix of P (position) $\qquad q \equiv P_q = P[1..q]$

$$\sigma(q,a) = \begin{cases} q+1, \text{ si } a=P[q+1] \\ max\{\, i \mid P_i \text{ is a suffix of } P_q a \,\} \text{ otherwise} \end{cases}$$

P = a b a a b



Invariant: P_q is the longest prefix of P which is a suffix of the prefix of T read so far

Pattern search in a sequence

Resulting algorithm:

1. *Pre-processing*: compute the automaton $O(m \cdot |A|)$ (time and space)
2. *Search*: run the automaton on the text $O(n \cdot \log(|A|))$

Goal: design an algorithm that does not depend on the alphabet size

Knuth-Morris-Pratt algorithm

```
T = … a b a b * * * * …
          = = = ≠
P =       b a b a b
```

What are possible shifts of the pattern?

Failure function:
$f(q)=max\{\ k \mid k<q$ et $P_k=P[1..k]$ is a suffix of $P_q\ \}$

P = a b a b a c a

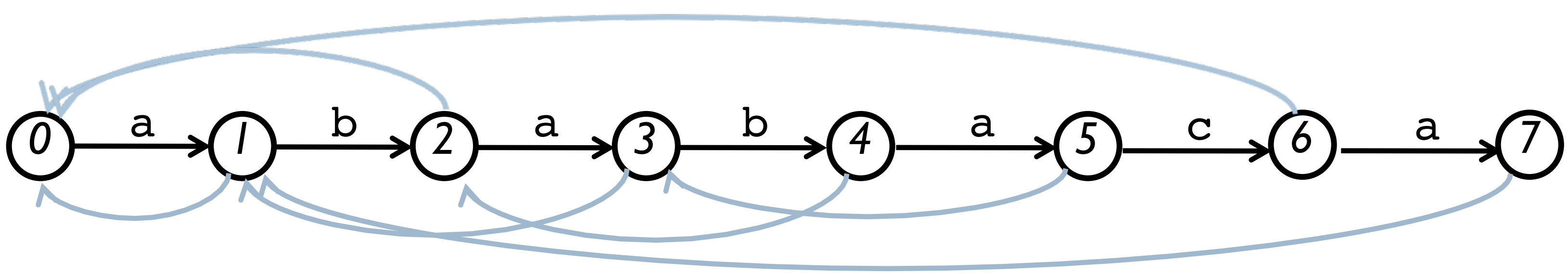
q	0	1	2	3	4	5	6	7
$f(q)$	−1	0	0	1	2	3	0	1

Knuth-Morris-Pratt algorithm

P = `a b a b a c a`

q	0	1	2	3	4	5	6	7
$f(q)$	−1	0	0	1	2	3	0	1

Knuth-Morris-Pratt "automaton"



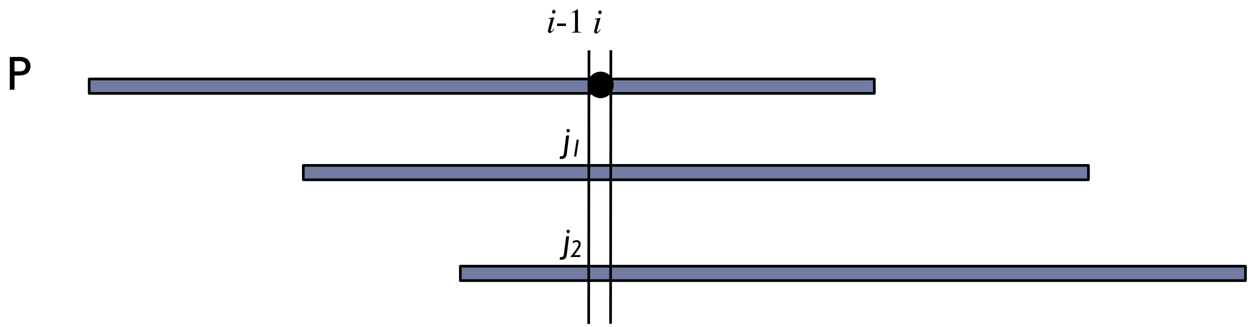
Knuth-Morris-Pratt algorithm

Once failure function f is computed …

KMP(T[1..n],f)
  j=0   /* pointer in P */
  for i=1 **to** n **do**
    while $j \geq 0$ **and** P[j+1]$\neq$T[i] **do**
      j=f(j) **endwhile**
    j=j+1
    if j==m **then**
      output(occurrence of P at position (i-m))
      j=f(j) **endif**
  endfor

Knuth-Morris-Pratt algorithm

How to compute the failure function



$j_1 = f(i-1)$

$j_2 = f(j_1)$

Shift the pattern against itself until P[j_q+1]=P[i]

Knuth-Morris-Pratt algorithm

Computation of the failure function

FF(P[1..m])

f[0]=-1

f[1]=0

k=0

for j=2 **to** m **do**

while $k \geq 0$ **and** P[k+1]≠P[j] **do**

k=f(k) **endwhile**

k=k+1

f(j)=k

endfor

Knuth-Morris-Pratt algorithm

Optimized verson (KMP vs MP)

```
T = … a b a * * * * * …
        = = ≠
P =   a b a b a c a
          a b a b a c a
```

⇐ useless shift

$h(3)=0$ h optimized failure function

q	0	1	2	3	4	5	6	7
$f(q)$	−1	0	0	1	2	3	0	1
$h(q)$	−1	0	0	0	0	3	0	1

Knuth-Morris-Pratt algorithm

Computation of the ***optimized*** failure function h

OFF(P[1..m])

  h[0]=-1

  h[1]=0

  k=0

  for j=2 **to** m **do**

    while $k \geq 0$ **and** P[k+1]≠P[j] **do**

      k=h(k) **endwhile**

    k=k+1

    ~~f(j)=k~~ **if** P[j] ≠P[k] **then** h(j)=h(k) **else** h(j)=k

  endfor

Knuth-Morris-Pratt algorithm

Difference with automaton approach: the algorithm can stay at the same position of the text during several steps

at most $\log_{\phi} m$ shifts at the same position, where $\phi=(1+\sqrt{5})/2$ is the golden ratio

Amortized time complexity

$O(n)$ for the search (KMP)
$O(m)$ for the pre-processing (FE)

History: Morris-Pratt (1970), Knuth-Morris-Pratt (1976), Matiyasevich (1971)

In practice: Boyer-Moore algorithm, $O(n/|A|)$ time on average

Exercise

- Give a linear-time algorithm to determine if a string T is a *cyclic rotation* (*conjugate*) of another string T '. That is, T=*uv* where T'=*vu*. For example,

  arc – car

  louche – chelou

  lenver (l’envers) – verlen (Verlan)

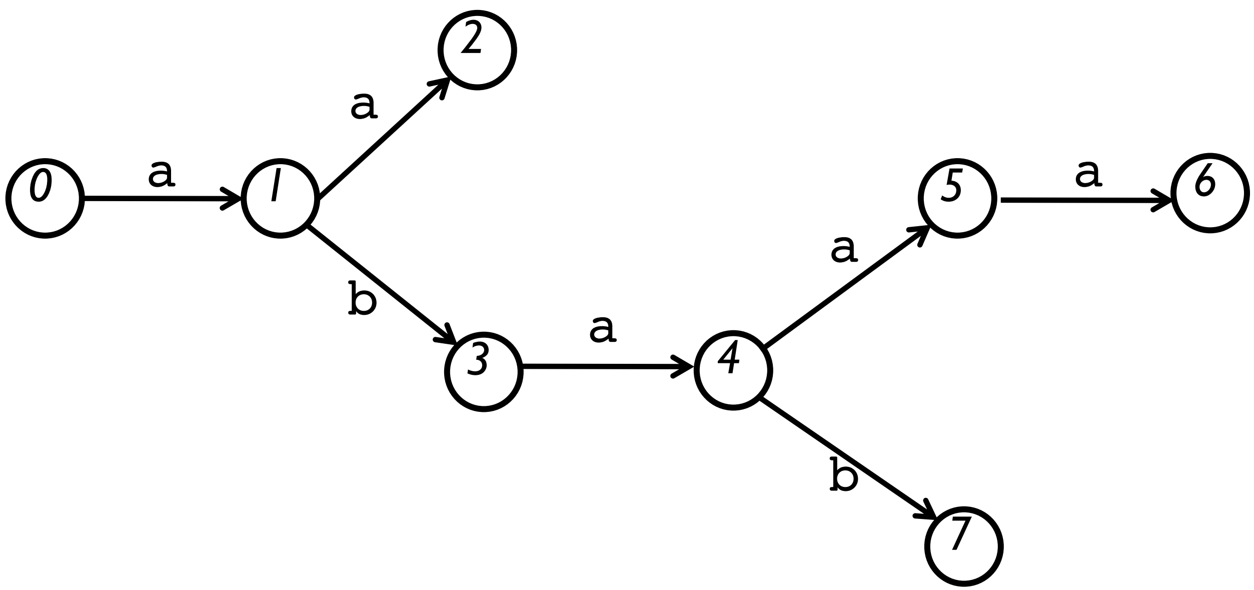
  окорок – рококо

Aho-Corasick algorithm (1984)

- Ideas of the Knuth-Morris-Pratt algorithm can be generalized to several patterns $\Rightarrow$ Aho-Corasick algorithm (1974)

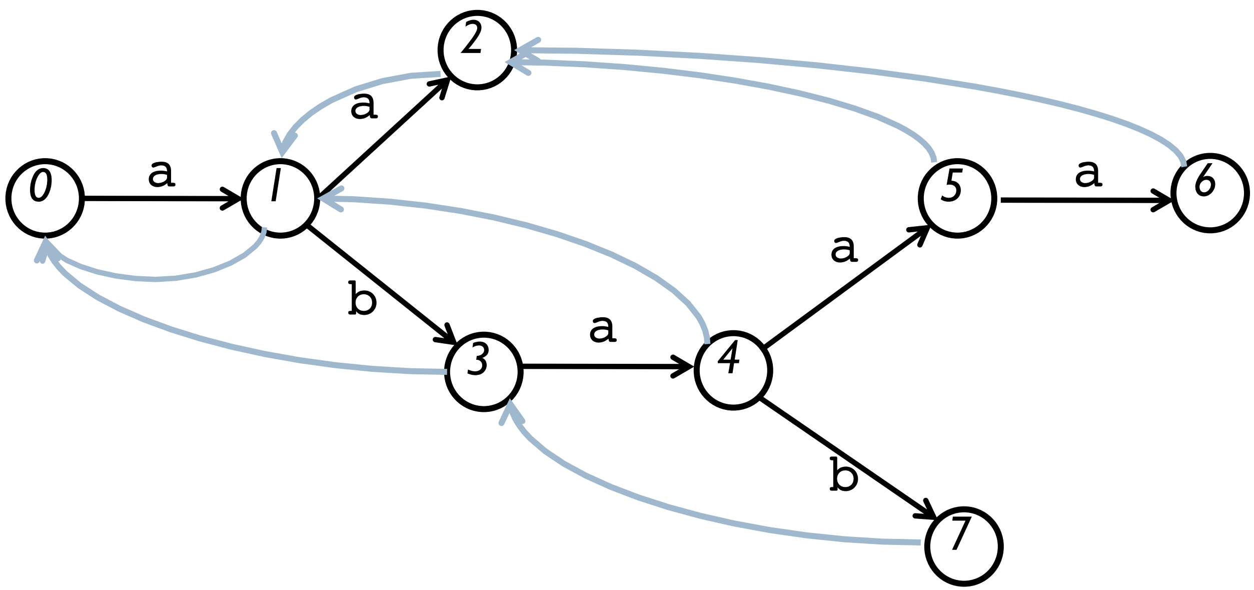
Aho-Corasick algorithm

- S={aa, abaaa, abab}
- Construct the *trie* of S



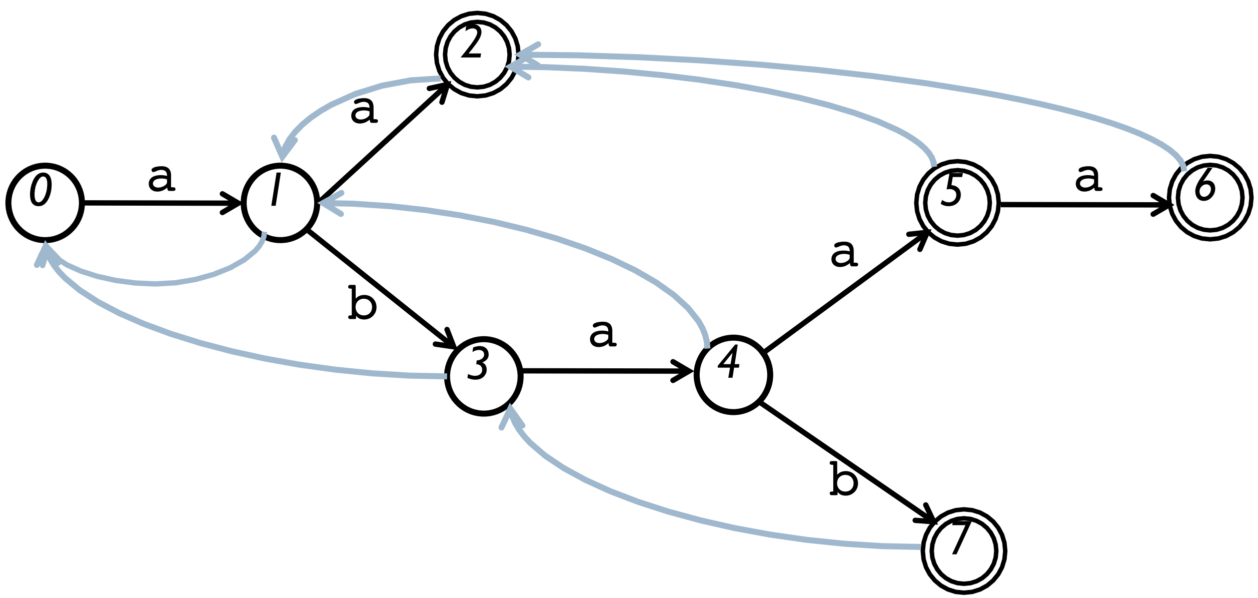
Aho-Corasick algorithm

- S={aa, abaaa, abab}
- Construct the *trie* of S, compute the failure function



Aho-Corasick algorithm

- S={aa, abaaa, abab}
- Construct the *trie* of S, compute the failure function, identify final states



Can be constructed in time O(m), where m is the **total** size of patterns in S

Karp-Rabin algorithm (1987)

- Use hashing for filtering!

Karp-Rabin algorithm (1987)

- Use hashing for filtering!
- let $A=\{0,1,2,3,4,5,6,7,8,9\}$
- encode pattern $P[1..m]$:

  $$p=P[1]\cdot 10^{m-1}+P[2]\cdot 10^{m-2}+\ldots+P[m-1]\cdot 10+P[m]$$
- p can be computed in time $O(m)$ with Horner's rule :

$$p=P[m]+10\cdot(P[m-1]+10\cdot(P[m-2]+\ldots+10\cdot(P[2]+10\cdot P[1])\ldots))$$
- idea:
  - iteratively compare p with encodings t_i of $T[i..i+m-1]$, for $i=1..n-m+1$
  - encoding of t_{i+1} computed from the encoding of t_i in constant time :

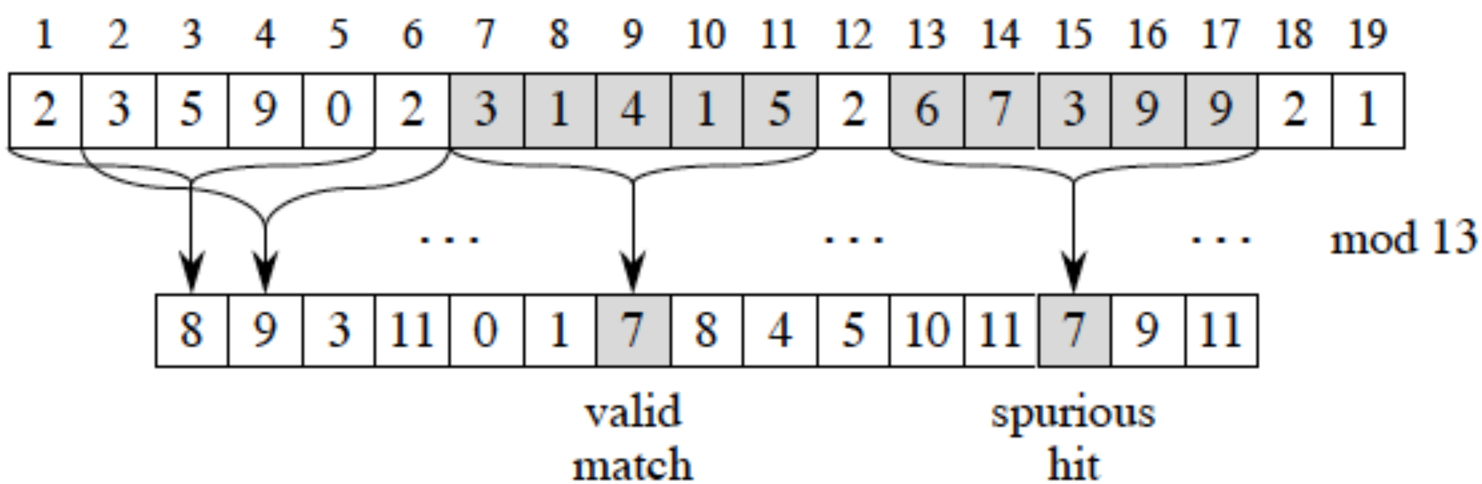
  $t_{i+1}=10\cdot(t_i-10^{m-1}\cdot T[i])+T[i+m]$ (assuming 10^{m-1} is pre-computed)
- *Problem*: encodings can be very large $\Rightarrow$ compute them modulo a prime number q

Karp-Rabin algorithm

- we then have

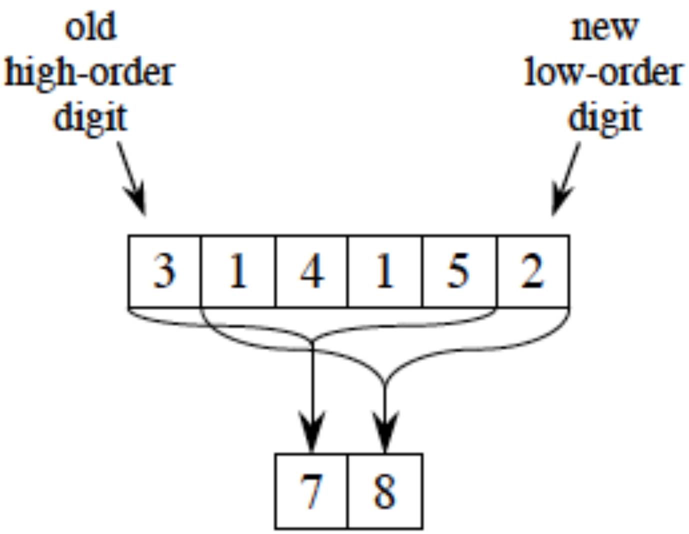
$t_{i+1}=10\cdot(t_i - h\cdot T[i])+T[i+m] \bmod q$, *where* $h=10^{m-1} \bmod q$

- *Problem*: we can have false positives!
- *Ex*: P=31415,   p=31415 mod 13 = 7



Karp-Rabin algorithm

- computing hash of t_i from hash of t_{i+1} (illustration):



old high-order digit   shift   new low-order digit

$$\begin{aligned} 14152 &\equiv (31415 - 3\cdot 10000)\cdot 10 + 2 \pmod{13} \\ &\equiv (7 - 3\cdot 3)\cdot 10 + 2 \pmod{13} \\ &\equiv 8 \pmod{13} \end{aligned}$$

- once a candidate (T[i..i+m-1] with the same hash) is found, we verify it by comparing P and T[i..i+m-1] letter-by-letter

Karp-Rabin hash function for strings

- $T = T[1..n]$
- Family of hash functions:
  - fix a large prime number p. In practice, $p = 2^{31} - 1$ for 32-bit and $p = 2^{61} - 1$ for 64-bit numbers
  - for $x \in [1..p-1]$, define $h_x(T) = (T[1] \cdot x^{n-1} + T[2] \cdot x^{n-2} + \cdots + T[n]) \bmod p$
- Family $\{h_x\}$ is n-universal, i.e. for two strings T and S of length n, $P[h_x(T) = h_x(S)] = n/p$, where probability is taken over a random choice of x
- $\Rightarrow$ excellent simple, practical and "almost universal" hash functions for strings