String Matching

Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών

Εθνικό Μετσόβιο Πολυτεχνείο

Problem Definition

- String : array of characters
  - Σ: alphabet
- Two strings are given:
  - a text T[1 … n]
  - a pattern P[1 … m]
- Problem: find the first substring that is the same as the pattern
- For every shift s: $T_s = T[s \ldots s+m-1]$
- Problem definition rephrased: find the smallest s such that $T_s = P$.
- In most cases m<<n

examples

- T="AMANAPLANACATACANAPANAMA"
  - P="CAN"
  - S=15
- T="AMANAPLANACATACANAPANAMA"
  - P="SPAM"
  - S=None

Applications

- Checking for Plagiarism in documents etc
- Bioinformatics and DNA sequencing
- Digital Forensics
- Spam Filter
- Search engines
- Intrusion Detection System

Almost Brute Force Algorithm

```
ALMOSTBRUTEFORCE(T[1..n], P[1..m]):
  for s ← 1 to n − m + 1
      equal ← TRUE
      i ← 1
      while equal and i ≤ m
          if T[s + i − 1] ≠ P[i]
              equal ← FALSE
          else
              i ← i + 1
      if equal
          return s
  return NONE
```

worst case:
Text: A..A n A's
Pattern A..AB m-1 A's
Complexity:
O((n-m)m)=O(nm)

Almost: break out of the inner loop at the first mismatch

Strings as Numbers

- Σ (alphabet) ={0,1,2,3,4,5,6,7,8,9}
  - p : Numerical Value of pattern P
  - T_s : Numerical Value of T_s

$$p = \sum_{i=1}^{m} 10^{m-i} \cdot P[i] \qquad t_s = \sum_{i=1}^{m} 10^{m-i} \cdot T[s+i-1]$$

- T= 3141592653589793<u>2384</u>626433832795028841971
  - m=4 T_{17} =2384
- Rephrasing problem definition: find the smallest s such that $p=t_s$

Strings as Numbers

- Compute p using Horner's Rule
  - time O(m)

$$p = P[m] + 10\left(P[m-1] + 10\left(P[m-2] + \cdots + 10\left(P[2] + 10 \cdot P[1]\right)\cdots\right)\right)$$

  - Computing t_s by the same way is useless (we get the same brute force algorithm)
- Compute t_{s+1} from t_s in constant time
  - subtract the most significant digit T[s] * 10^{m-1}
  - shift everything up by one digit
  - add the new least significant digit T[r+m]

$$t_{s+1} = 10\left(t_s - 10^{m-1} \cdot T[s]\right) + T[s+m]$$

  - T= 31415926535897932384626433832795028841971
  - t_s =2384 t_{s+1}=3846

Strings as Numbers

```
NumberSearch(T[1..n], P[1..m]):
    σ ← 10^(m−1)
    p ← 0
    t_1 ← 0
    for i ← 1 to m
        p ← 10 · p + P[i]
        t_1 ← 10 · t_1 + T[i]
    for s ← 1 to n − m + 1
        if p = t_s
            return s
        t_{s+1} ← 10 · (t_s − σ · T[s]) + T[s + m]
    return None
```

Complexity: $O(n)$?

Karp Rabin Fingerprinting (1981)

- Perform all arithmetic modulo some prime number q
  - q: 10*q fits into a standard integer variable (avoid long integer data type)
  - (p mod q) fingerprint of P (t_s mod q) fingerprint of T_s
- Compute (p mod q), (t_s mod q) in O(m). (Horner's rule)

$$p \bmod q = P[m] + \left(\cdots + \left(10 \cdot \left(P[2] + \left(10 \cdot P[1] \bmod q\right) \bmod q\right) \bmod q\right) \cdots \right)\right) \bmod q$$

- Given (t_s mod q) compute (t_{s+1} mod q) in constant time

$$t_{s+1} \bmod q = \left(10 \cdot \left(t_s - \left(\left(10^{m-1} \bmod q\right) \cdot T[s] \bmod q\right) \bmod q\right) \bmod q\right) + T[s+m] \bmod q$$

Karp Rabin Fingerprinting (1981)

- Two cases:
  - (p mod q) != (t_s mod q)   P != T_s
  - (p mod q) = (t_s mod q)   P = T_s ?
    - (if P != T_s) false match at shift s
    - test false match by *brute force* string comparison
    - F: number of false matches
    - Complexity *O(n+F*m)*
  - false match possibility 1/q
  - F=n/q
  - Complexity *O(n+n*m/q)*
  - if q>>m *O(n)*

Karp Rabin algorithm

```
KarpRabin(T[1..n], P[1..m]):
    q ← a random prime number between 2 and ⌈m² lg m⌉
    σ ← 10^(m−1) mod q
    p̃ ← 0
    t̃_1 ← 0
    for i ← 1 to m
        p̃ ← (10 · p̃ mod q) + P[i] mod q
        t̃_1 ← (10 · t̃_1 mod q) + T[i] mod q
    for s ← 1 to n − m + 1
        if p̃ = t̃_s
            if P = T_s        ⟨⟨brute-force O(m)-time comparison⟩⟩
                return s
        t̃_{s+1} ← (10 · (t̃_s − (σ · T[s] mod q) mod q) mod q) + T[s + m] mod q
    return None
```

Karp Rabin algorithm

- $\pi(u)$ the number of prime numbers less than u
- $\pi(m^2 \log m)$ possible values of q

- Lemma 1 $\pi(u) = \Theta(u/\log u)$
- Lemma 2 any integer x has at most $\lfloor \lg x \rfloor$ distinct prime divisors (if x has k prime divisors x>=2^k, since every prime number is bigger than 1)

- if there is a true match the algorithm ends early otherwise p!= t_s for every s
- if there is false match at s then q divides |p- t_s |

Karp Rabin algorithm

- $|p - t_s| < 10^m$ since both p, $t_s < 10^m$
- $|p - t_s|$ has at most O(m) prime divisors (lemma 2)
- q *is randomly chosen from a set of* $\pi(m^2 \log m)$ prime numbers
- probability of false match at shift s *O(1/m)*
- probability of false match at any shift *O(n/m)*
- Karp Rabin runs in *O(n)* **expected time**

Knuth Morris Pratt algorithm(1977)

- Redundant Comparisons (brute force algorithm)

  text = "HOPUSCOPUSABRA**B**RACADABRA"

  pattern = "ABRA**C**ADABRA"

  for s<11 algorithm fails from the very beginning

  for s=11 algorithm fails at fifth position

  for s=12 , s=13 algorithm fails

  for s=14, T[14]=P[4] match

  Once we've found a match for a text character, we never need to do another comparison with that text character again.(T[12], T[13])

  The next reasonable shift is the smallest value of s such that T[s .. i-1] which is a suffix of the previously-read text is also a proper prefix of the pattern (ABRA CAD ABRA)

example

text: “qwe**r**qwedqwrqwedqwedqwegwqedg”

pattern: “qwe**d**qweg” r!=d

q,w,e differ (there was a match from q do not expect match from w)

text: “**r**qwedqwrqwedqwedqwegwqedg”

pattern: “**q**wedqweg” r!=q

text: “qwedqw**r**qwedqwedqwegwqedg”

pattern: “qwedqw**e**g” r!=e

search pattern before “e” for prefix=suffix **qw** start pattern after qw from e

text: “**r**qwedqwedqwegwqedg”

pattern: “qw**e**dqweg” r!=e

pattern before e qw (q,w differ start pattern from scratch)

text: “rqwedqwedqwegwqedg”

pattern: “qwedqweg” r!=q advance text by 1

example

text: “qwedqwe**d**qwegwqedg”

pattern: “qwedqwe**g**” d!=g

search pattern for prefix-suffix before g: “qwe”

start pattern after qwe from d

text: “dqwegwqedg”

pattern: “qwedqweg”

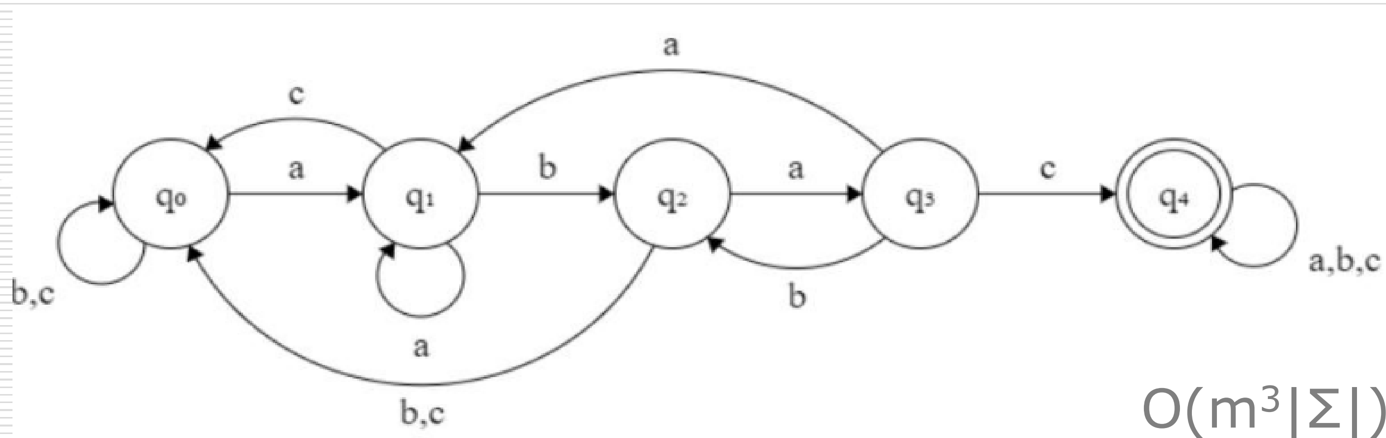
pattern found!

Deterministic Finite Automaton

Given an alphabet Σ={a,b,c} and a text T, how can we check if the string p="abac" belongs to the text?

A prefix or suffix of a string is *proper* if its length is less than that of the string.

From one state, the correct symbol takes us to the next state. With the wrong symbol we may stay in the same state or go to one of the previous states *never* to the next ones.



Knuth Morris Pratt algorithm

```
KnuthMorrisPratt(T[1..n], P[1..m]):
  j ← 1
  for i ← 1 to n
      while j > 0 and T[i] ≠ P[j]
          j ← fail[j]
      if j = m        ⟨⟨Found it!⟩⟩
          return i − m + 1
      j ← j + 1
  return None
```

Assume failure function known
worst case complexity: *O(n)*
At most n-1 failed comparisons (the number of time we decrease j can not exceed the number of time we increment j)

failure function

```
COMPUTEFAILURE(P[1..m]):
  j ← 0
  for i ← 1 to m
      fail[i] ← j        (*)
      while j > 0 and P[i] ≠ P[j]
          j ← fail[j]
      j ← j + 1
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$	e	k	e	k	e	d	e	k	e	k	e	d	e	k	e	k	e	k
fail	0	1	1	2	3	4	1	2	3	4	5	6	7	8	9	10	11	12

failure array example

pattern: "ekekedekekedekekek"

fail[i]: which is the longest suffix that is also prefix

```
ComputeFailure(P[1..m]):
  j ← 0
  for i ← 1 to m
      fail[i] ← j        (*)
      while j > 0 and P[i] ≠ P[j]
          j ← fail[j]
      j ← j + 1
```

- fail[0]=0 j=0
- i=1, j=0 fail[1]=j=0 j=1
- i=i+1=2, j=1 fail[2]=1   j>0 p[2]!=p[1], j=f(1)=0 j=1
- i=i+1=3, j=1 fail[3]=1   j>0 p[3]=p[1], j=j+1=2
- i=i+1=4, j=2 fail[4]=2   j>0 p[4]=p[2], j=j+1=3 j=3
- i=i+1=5, j=3 fail[5]=3   j>0 p[5]=p[3], j=j+1=4
- i=i+1=6, j=4, fail[6]=4   j>0 p[6]!=p[4], j=f(4)=2 longest prefix that is same as suffix has length 1, move j to the next position

  j=2 j>0 p[2]!=p[6] j=f[2]=1

  j=1 j>0 p[1]!=p[6] j=f[1]=0 // end of while loop
- j=0+1=1
- i=i+1=7, j=1, fail[7]=1 j>0 p[7]=p[1], j=j+1=2
- i=i+1=8, j=2, fail[8]=2 j>0 p[8]=p[2], j=j+1=3

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$	e	k	e	k	e	d	e	k	e	k	e	d	e	k	e	k	e	k
fail	0	1	1	2	3	4	1	2	3	4	5	6	7	8	9	10	11	12

failure array example

pattern: "ekekedekekedekekek"

fail[i]: which is the longest suffix that is also prefix

- i=i+1=8, j=2, fail[8]=j=2 j>0 p[8] =p[2], j=j+1=3
- i=i+1=9, j=3, fail[9]=j=3 j>0 p[9] =p[3], j=j+1=4
- i=i+1=10, j=4, fail[10]=j=4 j>0 p[10]=p[4], j=j+1=5
- i=i+1=11, j=5, fail[11]=j=5 j>0 p[11]=p[5], j=j+1=6
- i=i+1=12, j=6, fail[12]=j=6 j>0 p[12]=p[6], j=j+1=7
- i=i+1=13, j=7, fail[13]=j=7 j>0 p[13]=p[7], j=j+1=8
- i=i+1=14, j=8, fail[14]=j=8 j>0 p[14]=p[8], j=j+1=9
- i=i+1=15, j=9, fail[15]=j=9 j>0 p[15]=p[9], j=j+1=10
- i=i+1=16, j=10, fail[16]=j=10 j>0 p[16]=p[10], j=j+1=11
- i=i+1=17, j=11, fail[17]=j=11 j>0 p[17]=p[11], j=j+1=12
- i=i+1=18, j=11, fail[18]=j=12

0	1	2	3	4	5	6	7	8	9	10	11
$	A	B	R	A	C	A	D	A	B	R	A
fail	0	1	1	1	2	1	2	1	2	3	4

example

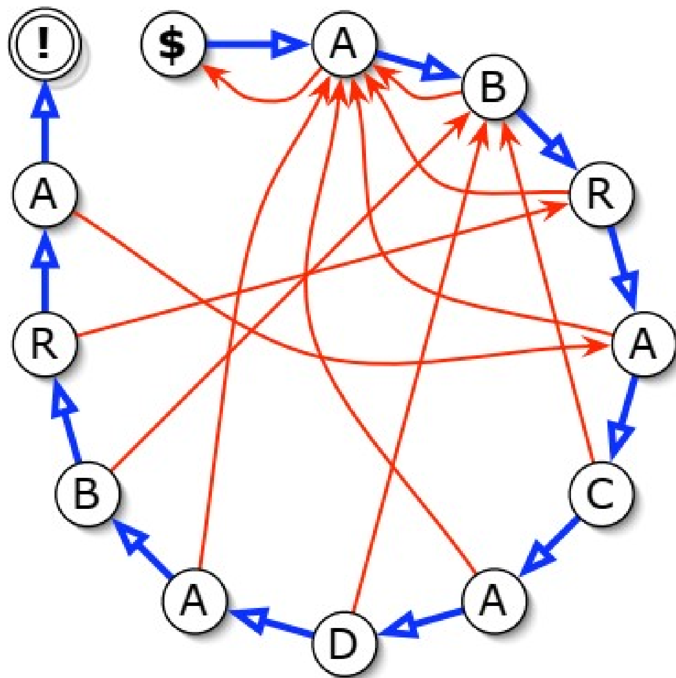
COMPUTEFAILURE($P[1..m]$):
- $j \leftarrow 0$
- for $i \leftarrow 1$ to m
  - $fail[i] \leftarrow j$ (∗)
  - while $j > 0$ and $P[i] \neq P[j]$
    - $j \leftarrow fail[j]$
  - $j \leftarrow j+1$

$j \leftarrow 0, i \leftarrow 1$	$\$^j$	A^i	B	R	A	C	A	D	A	B	R	X	...
$fail[i] \leftarrow j$		0											...
$j \leftarrow j+1, i \leftarrow i+1$	$	A^j	B^i	R	A	C	A	D	A	B	R	X	...
$fail[i] \leftarrow j$		0	1										...
$j \leftarrow fail[j]$	$\$^j$	A	B^i	R	A	C	A	D	A	B	R	X	...
$j \leftarrow j+1, i \leftarrow i+1$	$	A^j	B	R^i	A	C	A	D	A	B	R	X	...
$fail[i] \leftarrow j$		0	1	1									...
$j \leftarrow fail[j]$	$\$^j$	A	B	R^i	A	C	A	D	A	B	R	X	...
$j \leftarrow j+1, i \leftarrow i+1$	$	A^j	B	R	A^i	C	A	D	A	B	R	X	...
$fail[i] \leftarrow j$		0	1	1	1								...
$j \leftarrow j+1, i \leftarrow i+1$	$	A	B^j	R	A	C^i	A	D	A	B	R	X	...
$fail[i] \leftarrow j$		0	1	1	1	2							...
$j \leftarrow fail[j]$	$	A^j	B	R	A	C^i	A	D	A	B	R	X	...
$j \leftarrow fail[j]$	$\$^j$	A	B	R	A	C^i	A	D	A	B	R	X	...
$j \leftarrow j+1, i \leftarrow i+1$	$	A^j	B	R	A	C	A^i	D	A	B	R	X	...
$fail[i] \leftarrow j$		0	1	1	1	2	1						...
$j \leftarrow j+1, i \leftarrow i+1$	$	A	B^j	R	A	C	A	D^i	A	B	R	X	...
$fail[i] \leftarrow j$		0	1	1	1	2	1	2					...
$j \leftarrow fail[j]$	$	A^j	B	R	A	C	A	D^i	A	B	R	X	...
$j \leftarrow fail[j]$	$\$^j$	A	B	R	A	C	A	D^i	A	B	R	X	...
$j \leftarrow j+1, i \leftarrow i+1$	$	A^j	B	R	A	C	A	D	A^i	B	R	X	...
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	1				...
$j \leftarrow j+1, i \leftarrow i+1$	$	A	B^j	R	A	C	A	D	A	B^i	R	X	...
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	1	2			...
$j \leftarrow j+1, i \leftarrow i+1$	$	A	B	R^j	A	C	A	D	A	B	R^i	X	...
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	1	2	3		...
$j \leftarrow j+1, i \leftarrow i+1$	$	A	B	R	A^j	C	A	D	A	B	R	X^i	...
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	1	2	3	4	...
$j \leftarrow fail[j]$	$	A^j	B	R	A	C	A	D	A	B	R	X^i	...
$j \leftarrow fail[j]$	$\$^j$	A	B	R	A	C	A	D	A	B	R	X^i	...

0	1	2	3	4	5	6	7	8	9	10	11
$	A	B	R	A	C	A	D	A	B	R	A
fail	0	1	1	1	2	1	2	1	2	3	4

Finite State Machine

```
COMPUTEFAILURE(P[1..m]):
  j ← 0
  for i ← 1 to m
      fail[i] ← j        (*)
      while j > 0 and P[i] ≠ P[j]
          j ← fail[j]
      j ← j + 1
```



Finite State Machine

Labels: characters from the pattern

Edges: 2 outgoing **success**, failure

Iterate by 2 rules:

if T[i]=P[j] or current label ($) follow the success edge. Increment i.

if T[i]!=P[j] follow the failure edge. Do not change i.

(!) pattern found

Is it always possible to construct the whole graph? If the pattern is long?

The answer is:

failure function: fail[j] how far to shift after character mismatch (T[i]!=P[j])

failure function proof

```
ComputeFailure(P[1..m]):
  j ← 0
  for i ← 1 to m
      fail[i] ← j        (*)
      while j > 0 and P[i] ≠ P[j]
          j ← fail[j]
      j ← j + 1
```

- Is failure function computed correctly? Proof by Induction:
- Base case: fail[1]=0.
- Hypothesis: In line $(*)$ fail[1] through fail[i-1] are correct.
- Induction step: is fail[i] correct?
- After *i*-th iteration of line $(*)$ j=fail[i], so P[1..j-1] is the longest proper prefix of P[1..i-1] that is also a suffix.
- Definition of the iterated failure function $fail^c[j]$
- $fail^0[j]=j$, $fail^1[j]=fail(fail^0[j])=fail[j]$, $fail^2[j]=fail(fail^1[j])=fail(fail[j])$, $fail^c[j]=fail[fail^{c-1}[j]]$

$$fail^c[j] = fail[fail^{c-1}[j]] = \overbrace{fail[fail[\cdots[fail}^{c}[j]]\cdots]]$$

- Compute failure is a dynamic programming implementation of the following recursive implementation:

$$fail[i] = \begin{cases} 0 & \text{if } i = 0, \\ \max\limits_{c \geq 1} \left\{ fail^c[i-1]+1 \;\middle|\; P[i-1] = P[fail^c[i-1]] \right\} & \text{otherwise.} \end{cases}$$

Rabin Karp Algorithm vs Knuth-Morris-Pratt Algorithm

Rabin-Karp: Easier to implement when searching for multiple patterns in a text simultaneously, but involves understanding hashing.

Rabin-Karp: algorithm uses extra space to store hash value data.

KMP: Slightly more complex due to the preprocessing step, but very efficient and consistent for single-pattern matching.

KMP algorithm significantly reduces the time complexity from O(mn) to O(n), where m is the length of the pattern and n is the length of the text.

No Backtracking: Unlike other algorithms, KMP does not backtrack over the text, making it faster, especially for long texts.