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

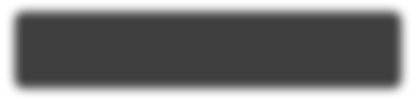
Both tend to be better for short patterns.

# Characters as digits

* Assume ∑ = {0,...,9}
* Then a string can be thought of as the decimal representation of a number:

## 427328

* In general, if |∑| = *d*, a string represents a number in base *d.*
* Let *p* = the number represented by query *P.*
* Let *ts* = the number represented by the |*P*| digits of *T* that start at position *s*.



*P*

occurs at position

*s*

of

*T*

⇔

*p*

=

*t*

*s*

.

# Computing p and ts

* Use Horner’s rule to compute *p* in time O(|*P*|=*m*):

*p = P*[*m*] + 10(*P*[*m*-1] + 10(P[*m*-2] + ... + 10(P[2] + 10P[1])...)

* Example: 427328 = (8+10(2+10(3+10(7+10(2 + 10 × 4)))))

“Left shift” by 1 digit

* *t*0 can be computed the same way in time O(|*P*|=*m*).
* *t*scan be computed from *ts*-1 in O(1) time:

*ts* = 10(*ts*-1 - 10*m*-1*T*[*s-1*]) + *T*[*s*+*m-1*]

shift left remove high- add next digit of

by 1 digit order digit T as the low-

order digit

# Rabin-Karp

Compute *p*.

Iteratively compute *ts*.

Output *s* when *ts* = *p*.

Problem: *p* and *ts* might be huge numbers.

Solution: compute everything modulo some prime *q.*

* If 10*q* is ≤ word size, then *p* mod *q* and *ts* mod *q* can be computed in a single word.
* If *p* occurs at *ts*, then *p* ≣ *ts* (mod *q*)

New problem: If *p* ≣ *ts* (mod *q*), it doesn’t necessarily mean there is a match at *s*.

New solution: if *p* ≣ *ts* (mod *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* ≣ *ts* (mod *q*1) *p* ≣ *ts* (mod *q*2) ⋮ *p* ≣ *ts* (mod *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*:

t1 t2 t3 t4 t5 t6 t7 t8 ... tm

M =

p

1

p

2

p

3

p

4

...

p

n

1

if

1

*P*

[1..

*i*

=

]

*T*

ending @

[

*j*

]

0

otherwise

*j*

1

’s in last row

will indicate *T:*where *P* matches *T*. *i*

*P:*

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

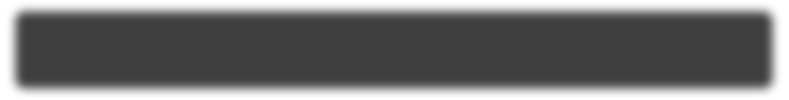
*M*[*i*,*j*] = (*P*[*i*] = *T*[*j*]) **and** (*P*[1..*i*-1] = *T*[ending @ *j*-1])

## *M*[*i*-1, *j*-1]

Computing M by columns

*M*[*i*,*j*] = *P*[*i*] = *T*[*j*] **and** *P*[1..*i*-1] = *T*[ending @ *j*-1]

## *M*[*i*-1, *j*-1]



**Def.**

*U*

*P*

(

*x*

)

= |

*P*

|-bit vector where

*i*

th

entry is 1 if

*P*

[

*i*

]

=

*x*

, 0 otherwise.

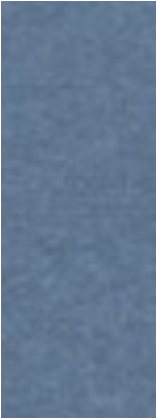
Compute columns of *M* left to right:

|  |  |  |
| --- | --- | --- |
| *j*th column of *M* | 1 where *P*[*i*] = *T*[*j*] | previous column of  *M*, shifted down by 1  (prepended with a 1) |

*M*[*•*,*j*] = *UP*(*T*[*j*]) **&** (1; *M*[•,*j*-1])

first entry always 1

=**&**because conditionred is empty



0



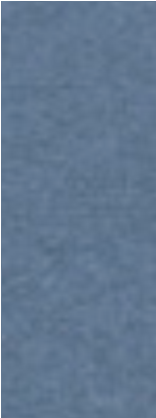
0



0



1



0



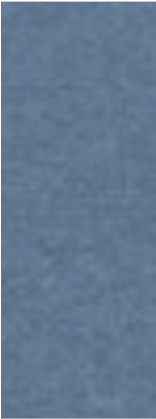
0



1



1



1



1



0



1

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*| × |*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

*j*

|  |  |  |
| --- | --- | --- |
|  |  | M0 |

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

|  |  |  |
| --- | --- | --- |
|  |  | M1 |

*Ml*[*i,j*] = *i*th prefix of *P* matches suffix ending at *j* of *T* **with ≤ *l* mismatches.**

|  |  |  |
| --- | --- | --- |
|  |  | M2 |

|  |  |
| --- | --- |
| ≤ *l-1* mismatches ≤ *l*-1 mismatches  *Ml*[*j*] = *Ml-1*[*j*] **or** (bs(*Ml*(*j*-1)) **and** *U*(*T*(*j*))) **or** *Ml*-1[*j*-1] | .  .  . |

|  |  |  |
| --- | --- | --- |
|  |  | M*k* |

*i*th prefix of *P* *i*th prefix of *P*

matches string matches string

ending at *j* with ending at *j*-1 with

*i*-1 characters of *P* match with ≤ *l* mismatches and *j*th character matches.

# 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.