

# String Algorithms

- Sequence and string are synonymous
  - Finite number of letters<sup>1</sup> from an alphabet, written contiguously from left to right  $S_{i..j}$

ACTGGTCA

- Subsequence

- an ordered subset obtained by removing letters from a sequence

ACTGGTCA  
  ↓  ↓  ↓  ↓  ↓  
  ATGGTA

(Both C's are removed)

- Substring

- Consecutive and complete string of letters found in the target string

ACTGGTCA  
  CTGGT

<sup>1</sup>Sometimes called symbols or characters

# Prefix and Suffix

SLITHY      N=6

*For*  $S_{1 \leq i \leq j \leq n}$

*Prefix* :  $S_{1 \dots j}$

S, SL, SLI, SLIT, SLITH, SLITHY

*Suffix* :  $S_{i \dots n}$

Y, HY, THY, ITHY, LITHY, SLITHY

# String Algorithms

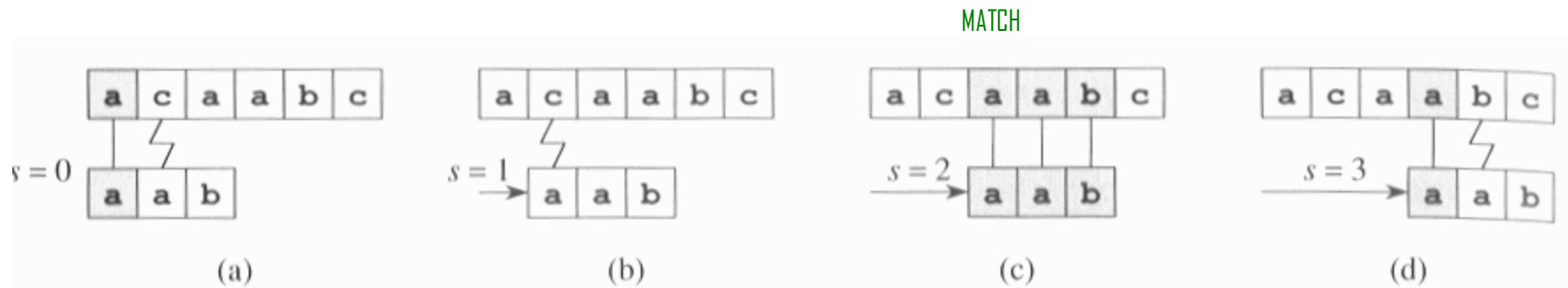
- Algorithms for String Matching
  - No preprocessing
    - Naïve
    - Boyer-Moore
  - Preprocessing of target string
    - Aho-Corasick
  - Preprocessing of query string, target string or fixed DB
    - Rabin-Karp
    - Suffix Tree
- Algorithms for String Alignment
  - No preprocessing
    - Needleman Wunsch
    - Smith Waterman
  - Preprocessing of 'fixed' Database
    - Complicated heuristics such as BLAST, FASTA

# STRING MATCHING

Looking for an EXACT substring match

# Naïve algorithm

## Operates in quadratic time



Complexity  $\sim mn$

# Boyer-Moore

- ‘Smarter’. Operates in linear time
- Pattern string  $s$  doesn’t ‘walk’ the target string  $t$ ; it ‘skips along’ the target string
- At each iteration, it chooses the better of:
  - Jump the distance derived from the bad character heuristic

Or

- Jump the distance derived from the good suffix heuristic

# Example

- **In the dictionary** . (296 characters, including spaces, excluding punctuation)

Now o'er the one halfworld

Nature seems dead, and wicked dreams abuse

The curtain'd sleep; witchcraft celebrates

Pale Hecate's offerings, and wither'd murder,

Alarum'd by his sentinel, the wolf,

Whose howl's his watch, thus with his stealthy pace

With Tarquin's ravishing strides, towards his design

Moves like a ghost

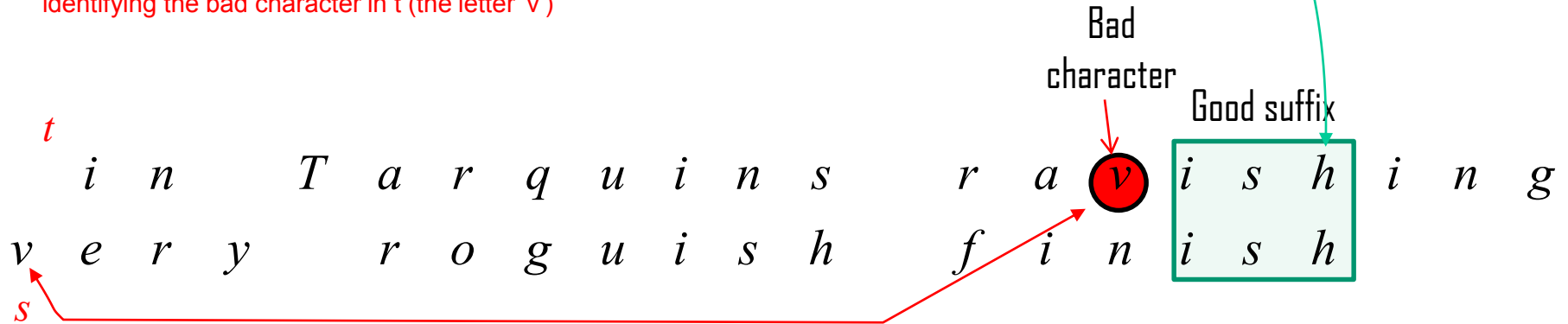
- **Look up** (19 characters, including spaces, excluding punctuation)

Very roguish finish

# Bad Character Heuristic

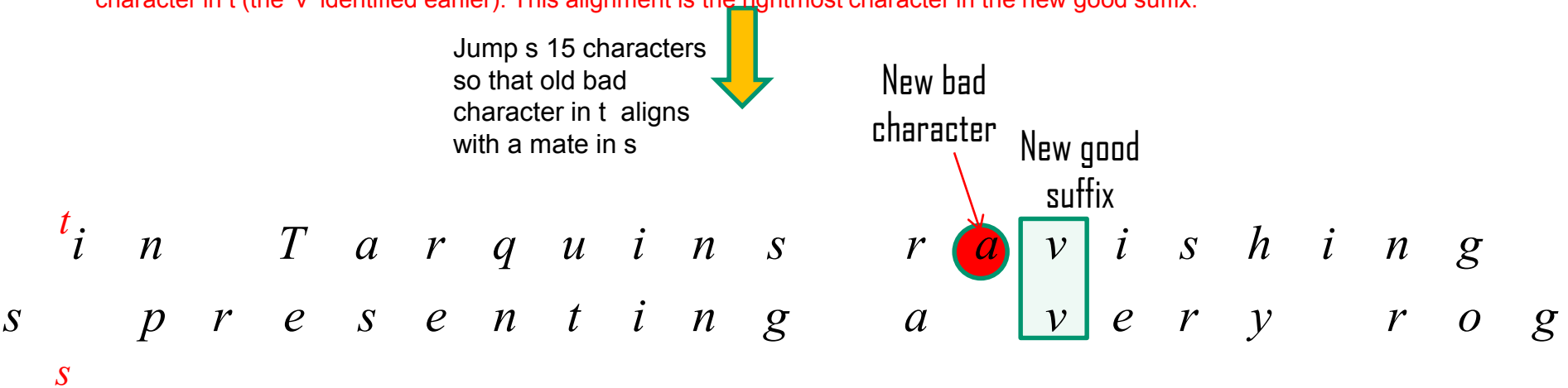
Use naïve algorithm to align last character in *s* with corresponding character in *t*. (this is the letter 'h')

Then, extend match leftwards for increasing suffixes of *both strings* (letters 's', then 'i'). Stop with a mismatch, identifying the bad character in *t* (the letter 'v')



Slide *s* to the right to align the rightmost occurrence of a same bad character in *s* (find a letter 'v') with a the bad character in *t* (the 'v' identified earlier). This alignment is the rightmost character in the new good suffix.

Jump *s* 15 characters  
so that old bad  
character in *t* aligns  
with a mate in *s*



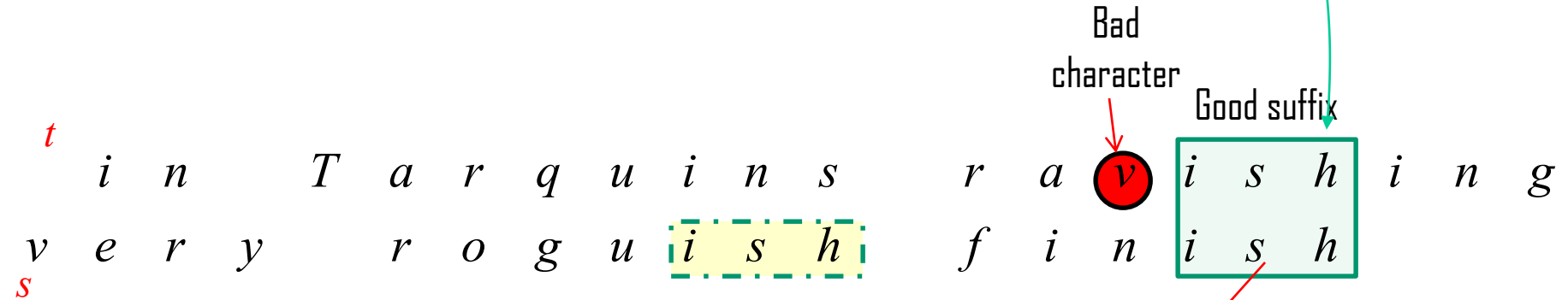
Note that we don't use this new good suffix just yet; first we must look at the results from the good suffix heuristic (next slide)



# GOOD SUFFIX Heuristic

As before, use naïve algorithm to align last character in  $s$  with corresponding character in  $t$ . (this is the letter 'h')

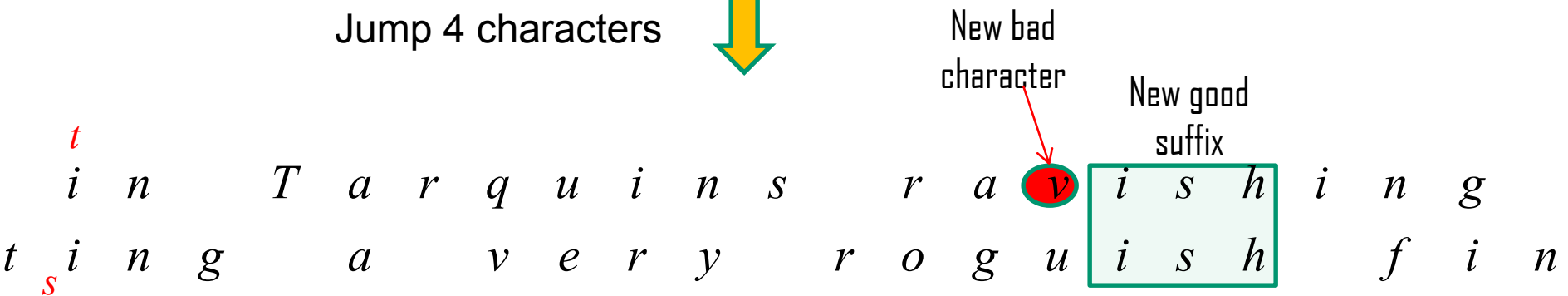
Then, extend match leftwards for increasing suffixes of *both strings* (letters 's', then 'i'). Stop with a mismatch, identifying the bad character in  $t$  (the letter 'v')



Find the rightmost suffix in  $s$  to the left of the original good suffix in  $s$  that matches this good suffix

Slide  $s$  to align it with a corresponding suffix in  $t$ .

Jump 4 characters




Choose the larger jump!

# AHO-CORASICK

## 2 Phases

- Preprocess: Build a finite automaton based on the query strings (quadratic time)
- Use the target string as input to the automaton (linear time)

# The Constructed Automaton for the string 'CATCH'

<i>input</i> →		<i>A</i>	<i>C</i>	<i>H</i>	<i>T</i>	
<i>state</i> ↓						
$q_0$		$q_0$	$q_1$	$q_0$	$q_0$	
$q_1$		$q_2$	$q_0$	$q_0$	$q_0$	
$q_2$		$q_0$	$q_0$	$q_0$	$q_3$	$q_5 = \text{stop}$
$q_3$		$q_0$	$q_4$	$q_0$	$q_0$	
$q_4$		$q_0$	$q_0$		$q_0$	

$q_0 = \text{start}$

Note that this abstract machine is specifically an acceptor for the word 'catch'. The machine simply halts if 'catch' is input, otherwise it remains in  $q_0$

# Deterministic Finite Automaton

Q: Could this scheme work for

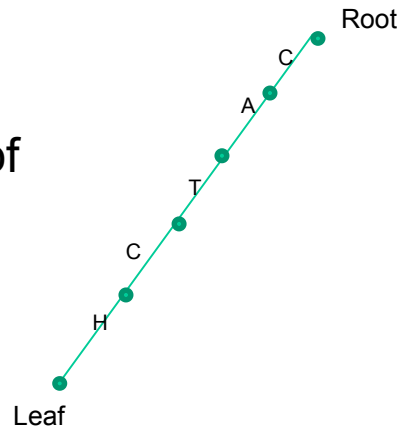
MISSISSIPPI ?

The word "MISSISSIPPI" is displayed in a large, black, sans-serif font. A green rectangular box highlights the first 'S' character, and a red rectangular box highlights the second 'S' character. The boxes overlap slightly at the top and bottom edges. A question mark follows the word.

# Aho-Corasick

The deterministic automaton can be represented by a rooted prefix tree. Each edge is a character in the target string and each node represents a prefix formed by the edges along the path from the root to that node. When there is a match the query string maps to a leaf of the tree.

Here is a prefix tree of our simple example

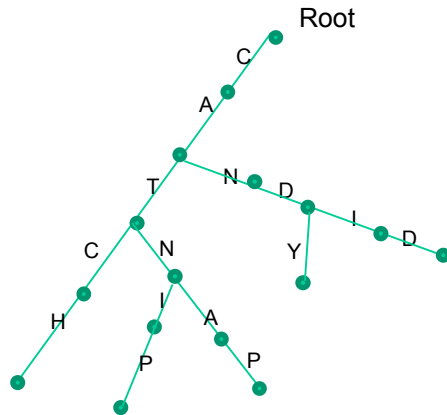


# Generalized Aho-Corasick

Many patterns can be represented on a single prefix tree (Keyword Tree)

Such trees can be created for entire dictionaries in linear time in a preprocessing step. The tree can then be traversed with a query word as needed

Not shown here are the failure links



# Pro's and Con's of String Matching Algorithms Thus Far

- Pro
  - No false negatives, no false positives
- Con
  - Significant overhead/slowdown by comparing characters
  - If characters are converted to numbers, the numbers can become quite large
    - Large arithmetic computational overhead (slowdown)
    - Very large numbers can blow RAM





# String Searching with 'Fingerprints'

## The Rabin-Karp Algorithm

- Pro
  - Because the algorithm uses modular arithmetic, numbers and intermediate calculations are constrained, so...
    - Very fast
    - No threat to RAM limits
  - No false negatives
  - The probability of a false positive is extremely low
- Con
  - False positives are nonetheless possible
    - If the probability of a false positive is not deemed low enough, or any false positive whatsoever is unacceptable, then some small overhead is required for checking

# Essence of the Algorithm

- Every possible substring in the target of the same length as the query string is identified.
- The query string and every target substring so identified is converted to an integer
- Each integer is mapped into its remainder modulo  $p$ , where  $p$  is a carefully chosen prime number
- The query string (new) integer is compared to every possible target substring (new) integer in the target string.
  - When the query and target numbers are not the same, there is no match
  - When the query and target numbers are the same, there is a very high probability of a match

# RABIN-KARP SEARCH

Suppose we have an alphabet consisting only of the letters AFPQRSTVWY, coded by decimal digits. These digits represent the 10 amino acids found in life forms in Star System TGR-75 (our own planet has 20 naturally occurring amino acids)

Amino Acid		Letter	Numeric Code
ALANINE	Ala	A	0
PHENYLALANINE	Phe	F	9
PROLINE	Pro	P	8
GLUTAMINE	Gln	Q	7
ARGININE	Arg	R	6
SERINE	Ser	S	5
THREONINE	Thr	T	4
VALINE	Val	V	3
TRYPTOPHAN	Trp	W	2
TYROSINE	Tyr	Y	1

These amino acids combine in a linear chain to form proteins. There are thousands of proteins, each with a unique sequence of amino acids in the chain, some chains very long (1000's), some short (5). All variations of these proteins, and all proteins, are all ordered sequentially in a single searchable database, totaling seven million entries. For convenience and storage economy, the numeric code is used instead of the alphabet letter.

# The Computational Problem:

We have just synthesized a new protein (or so we think). The sequence of amino acids is 5 long: Val-Tyr-The-Tyr-Ser (code is 31415).

We wish to determine whether this protein (sequence of amino acids) exists already. We agree that it is better to work with the AA codes, rather than the names, so we are searching the DB for at least one instance of 31415 as our target integer

# Computational Solution

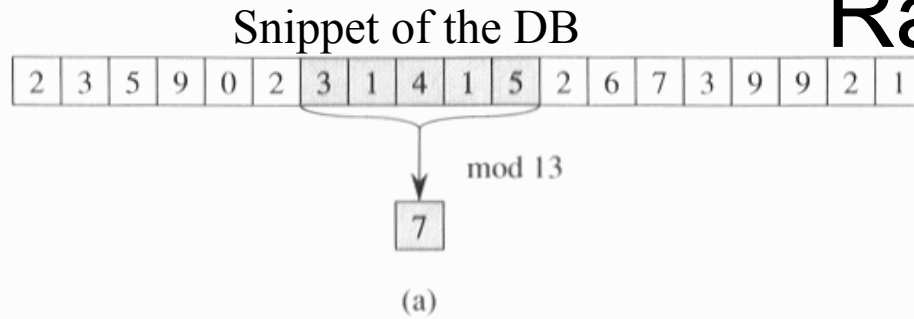
- Strategy 1: Combine each successive 5 digits in the target into a 5 digit integer, then walk the set of target integers, seeking the query integer
  - Advantage: No false positives
  - Disadvantage: Could be faster: Must walk the DB and make a complex comparison each time (polynomial)
- Strategy 2: Preprocess the database, building a hash table of hash values, then seek the hash value matching the hash value of the query
  - Advantage: Fast. Preprocess only once in linear time, search with multiple queries in linear time. Smaller database to search
  - Disadvantage: Must deal with hash collisions (false positives), although still fast

# Rabin-Karp

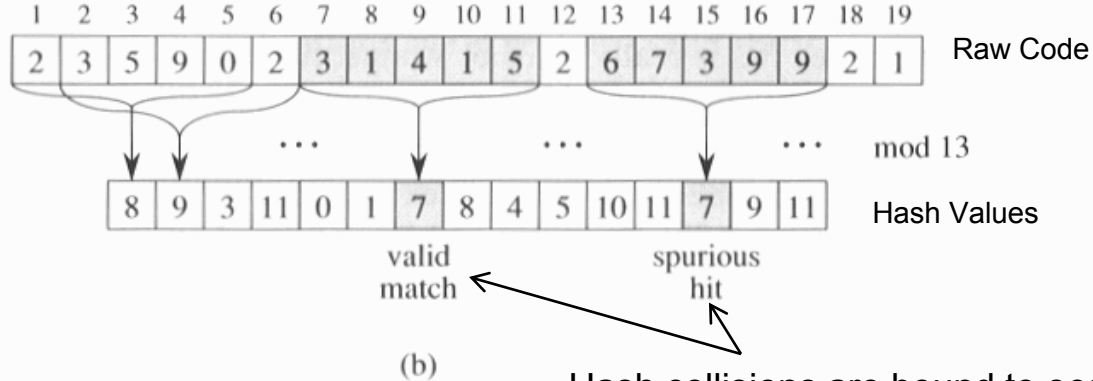
Desired string is  
31415

Hash function is  
 $X_{\text{mod } p} \quad p=13$

Desired hash value  
is 7



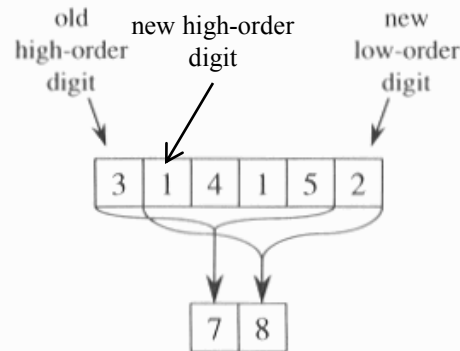
Computing the hash  
value while walking the  
target (strategy #1) or  
preprocess DB building a  
hash table (strategy #2)



Hash collisions are bound to occur

Efficiently  
(linearly)  
generating a  
new hash value:

- Take away the high order digit
- shift everything left with zero-fill
- add value of new right target digit
- Apply hash function



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

(c)

# About Efficiency

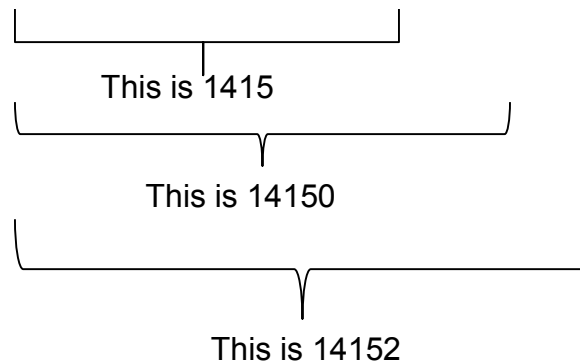
The efficiency is that working in modulo arithmetic saves combining 5 digits into a 5 digit integer, then working with a decimal shift and finding a remainder. Here is a naïve way to slide along

Let's slide along the DB from 31415 to the next letter (2), ending in 14152

$$31415 = 30,000 + 1415$$

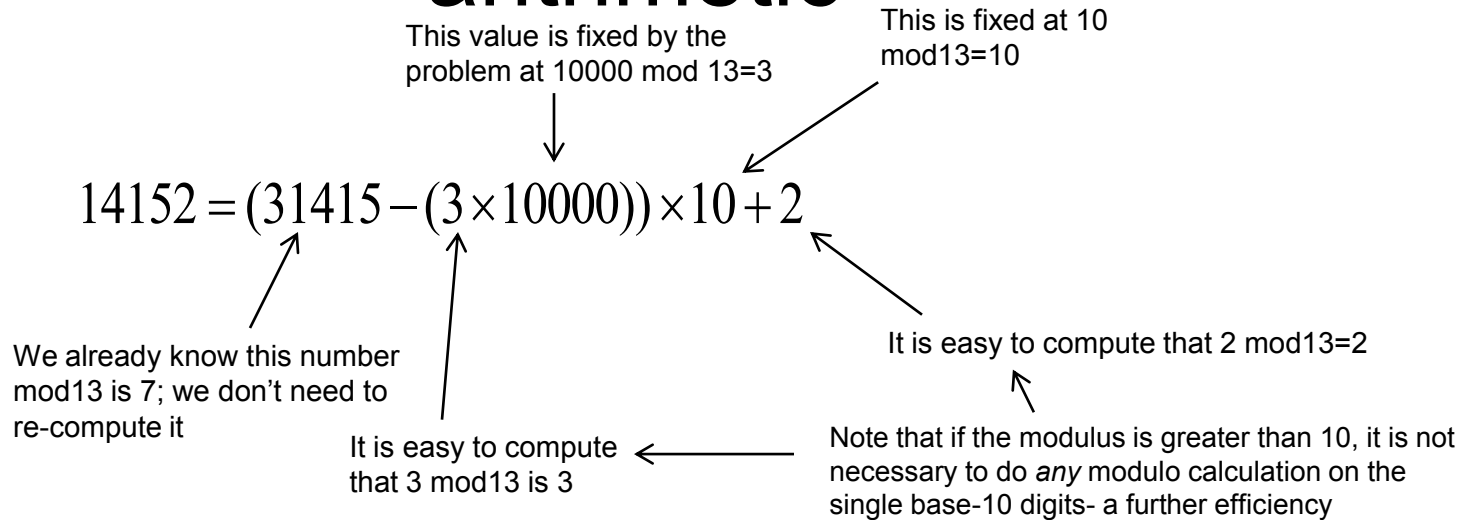
So, we need to get to 14150, then add 2

$$14152 = (31415 - (3 \times 10000)) \times 10 + 2$$



$$14152 \bmod 13 = 8$$

# Working efficiently in modulo arithmetic



By theorem:  $[xy]_k \equiv [x]_k [y]_k$

$$[3 \times 10000]_{\bmod 13} \equiv [3]_{\bmod 13} \times [10000]_{\bmod 13}$$

$$\equiv 3_{\bmod 13} \times 3_{\bmod 13} = 9$$

So, walking the DB in modulo arithmetic is highly efficient

$$\begin{aligned}
 14152 \bmod 13 &= [(7 - (3 \times 3)) \times 10 + 2] \bmod 13 \\
 &= [(7 - 9) \times 10 + 2] \bmod 13 \\
 &= [-18] \bmod 13 = 8
 \end{aligned}$$



# What modulus?

Answer: A prime number!

- A prime modulus limits the probability of a false positive
- Prime moduli in a modular hashing function enhance diffusion and prevent clumping of false positives (hash collisions) in a hash table

# Probability of false positives\* .....

$n$  is the length of the query

$m$  is the length of the target

*Let  $\pi(x)$  be the number of prime numbers  $\leq$  any positive integer  $x$ , and  $\pi(mn)$  likewise*

*Ensure that  $n \cdot m \geq 29$*

*Choose a prime  $p \leq x$  at random for any positive integer  $x$*

*Theorem :  $\text{prob}(\text{spurious hit}) \leq \frac{\pi(n \cdot m)}{\pi(x)}$*

\*Development follows Gusfield

# False Positives

We need to pick an  $x$  such that a random prime number  $\leq x$  is neither too big (slows down things) nor too small (too many false positives)

Without getting into the nuances of number theory, let us make an empirical\* choice and let  $x$  be  $nm^2$ .

In that case, the upper limit on the probability of a false positive is  $2.53/m$ .

So, for our target snippet of length 19 and our query of length 5, the probability of a false positive is  $2.53/19$ , or  $\sim 13\%$ . Not all that great, but we have a tiny database

But consider our entire Star System TGR-75 database. If we were to pick any prime number  $\leq 5 \times (7,000,000)^2$  at random as our modulus, the probability of a false positive with that choice would be  $3.614 \times 10^{-7}$ .

\*If interested, perhaps as a final project, you might explore the prime number counting functions of Gauss and Chebychev

# Suffix Trees

- Preprocessing of the target (Tree building: linear)
- Linear Search time
- Can generalize to multiple targets
  - Allows other functions besides exact matching
  - Longest Common Substring
  - Wildcards

# Suffix Trees : An example

***And, has thou slain the Jabberwock?  
Come to my arms, my beamish boy!  
O frabjous day! Callooh! Callay!  
He chortled in his joy.***

***-Lewis Carroll***

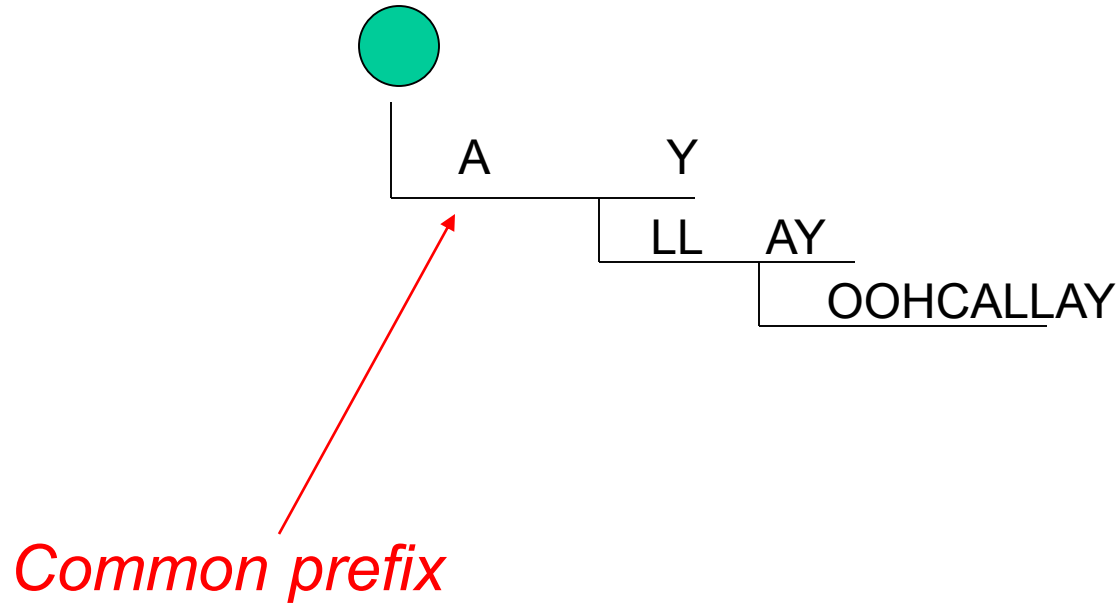
# Suffixes of *CalloohCallay*

CalloohCallay	Callay
alloohCallay	allay
lloohCallay	llay
loohCallay	lay
oohCallay	ay
ohCallay	y
hCallay	

# Suffixes of *CalloohCally*: ordered

ay	lay
allay	llay
alloohCally	lloohCally
Callay	loohCally
CalloohCally	ohCally
hCally	oohCally
y	

# Constructing the Tree





# Suffix Tree for 'CalloohCallay'

