6 String Matching – What's behind Ctrl+F?

17 November 2025

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2025–26)
Philipps-Universität Marburg
version 2025-09-28 17:50 H

Learning Outcomes

Unit 6: ***String Matching***

1. Know and use typical notions for *strings* (substring, prefix, suffix, etc.).
2. Understand principles and implementation of the *KMP*, *BM*, and *RK* algorithms.
3. Know the *performance characteristics* of the KMP, BM, and RK algorithms.
4. Be able to solve simple *stringology problems* using the *KMP failure function*.

Outline

6 String Matching

6.1 String Notation

Ubiquitous strings

string = sequence of characters

- universal data type for . . . everything!
  - natural language texts
  - programs (source code)
  - websites
  - XML documents
  - DNA sequences
  - bitstrings
  - . . . a computer's memory $\rightsquigarrow$ ultimately any data is a string

$\rightsquigarrow$ many different tasks and algorithms

- This unit: finding (exact) **occurrences of a pattern** text.
  - Ctrl+F
  - grep
  - computer forensics (e. g. find signature of file on disk)
  - virus scanner

- basis for many advanced applications

Notations

- *alphabet* Σ: finite set of allowed **characters**; $\sigma = |\Sigma|$ *"a string over alphabet Σ"*
  - letters (Latin, Greek, Arabic, Cyrillic, Asian scripts, ...)
  - "what you can type on a keyboard", Unicode characters
    comprehensive standard character set including emoji and all known symbols
  - $\{0,1\}$; nucleotides $\{A, C, G, T\}$; ...
- $\Sigma^n = \Sigma \times \cdots \times \Sigma$: strings of **length** $n \in \mathbb{N}_0$ (n-tuples)
- $\Sigma^\star = \bigcup_{n \ge 0} \Sigma^n$: set of **all** (finite) strings over Σ
- $\Sigma^+ = \bigcup_{n \ge 1} \Sigma^n$: set of **all** (finite) **nonempty** strings over Σ
- $\varepsilon \in \Sigma^0$: the *empty* string (same for all alphabets)
- zero-based (like arrays)!
  for $S \in \Sigma^n$, write $S[i]$ (other sources: S_i) for i**th** character $(0 \le i < n)$
- for $S, T \in \Sigma^\star$, write $ST = S \cdot T$ for **concatenation** of S and T
- for $S \in \Sigma^n$, write $S[i..j]$ or $S_{i,j}$ for the **substring** $S[i] \cdot S[i+1] \cdots S[j]$ $(0 \le i \le j < n)$
  - $S[0..j]$ is a **prefix** of S; $S[i..n-1]$ is a **suffix** of S
  - $S[i..j) = S[i..j-1]$ (endpoint exclusive) $\rightsquigarrow$ $S = S[0..n)$

String matching – Definition

Search for a string (pattern) in a large body of text

- **Input:**
  - $T \in \Sigma^n$: The *text* (haystack) being searched within
  - $P \in \Sigma^m$: The *pattern* (needle) being searched for; typically $n \gg m$
- **Output:**
  - the *first occurrence (match)* of P in T: $\min\{i \in [0..n-m) : T[i..i+m) = P\}$
  - or `NO_MATCH` if there is no such i ("P does not occur in T")
- Variant: Find **all** occurrences of P in T.
  $\rightsquigarrow$ Can do that iteratively (update T to $T[i+1..n)$ after match at i)
- **Example:**
  - T = "`Where is he?`"
  - P_1 = "`he`" $\rightsquigarrow$ $i = 1$
  - P_2 = "`who`" $\rightsquigarrow$ `NO_MATCH`
- string matching is implemented in Java in `String.indexOf`, in Python as `str.find`

6.2 Brute Force

Abstract idea of algorithms

String matching algorithms typically use *guesses* and *checks*:

- A **guess** is a position i such that P might start at $T[i]$.
  Possible guesses (initially) are $0 \le i \le n - m$.

- A **check** of a guess is a comparison of $T[i + j]$ to $P[j]$.

- Note: need all m checks to verify a single *correct* guess i,
  but it may take (many) fewer checks to recognize an *incorrect* guess.

- Cost measure: #character comparisons

$\rightsquigarrow$ #checks $\le n \cdot m$ (number of possible checks)

Brute-force method

```
procedure bruteForceSM(T[0..n), P[0..m)):
    for i := 0, ..., n − m − 1 do
        for j := 0, ..., m − 1 do
            if T[i + j] ≠ P[j] then break inner loop
        if j == m then return i
    return NO_MATCH
```

- try all guesses i
- check each guess (left to right); stop early on mismatch
- essentially the implementation in Java!

- **Example:**
  T = abbbababbab
  P = abba

⇝ 15 char cmps
(vs $n \cdot m = 44$)
not too bad!

a	b	b	b	a	b	a	b	b	a	b
a	b	b	a							
	a									
		a								
			a							
				a	b	b				
					a					
						a	b	b	a	

Brute-force method – Discussion

Brute-force method can be good enough

- typically works well for natural language text
- also for random strings

but: can be as bad as it gets!

a	a	a	a	a	a	a	a	a	a	a
a	a	a	b							
	a	a	a	b						
		a	a	a	b					
			a	a	a	b				
				a	a	a	b			
					a	a	a	b		
						a	a	a	b	
							a	a	a	b

- Worst possible input: $P = a^{m-1}b$, $T = a^n$
- Worst-case performance: $(n - m + 1) \cdot m$

$\rightsquigarrow$ for $m \le n/2$ that is $\Theta(mn)$

- Bad input: lots of self-similarity in T! $\rightsquigarrow$ can we exploit that?
- brute force does 'obviously' stupid repetitive comparisons $\rightsquigarrow$ can we avoid that?

Roadmap

- **Approach 1** (this week): Use *preprocessing* on the **pattern** P to eliminate guesses
  (avoid 'obvious' redundant work)
  - Deterministic finite automata (**DFA**)
  - **Knuth-Morris-Pratt** algorithm
  - **Boyer-Moore** algorithm
  - **Rabin-Karp** algorithm

- **Approach 2** ($\rightsquigarrow$ Unit 13): Do *preprocessing* on the **text** T
  Can find matches in time *independent of text size(!)*
  - **inverted indices**
  - **Suffix trees**
  - **Suffix arrays**

6.3 String Matching with Finite Automata

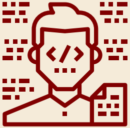
Theoretical Computer Science to the rescue!

- string matching $=$ deciding whether $T \in \Sigma^\star \cdot P \cdot \Sigma^\star$
- $\Sigma^\star \cdot P \cdot \Sigma^\star$ is *regular* formal language

$\rightsquigarrow$ $\exists$ *deterministic finite automaton* (DFA) to recognize $\Sigma^\star \cdot P \cdot \Sigma^\star$

$\rightsquigarrow$ can check for occurrence of P in $|T| = n$ steps!

Job done!



`WTF!?`

We are not quite done yet.

- (Problem 0: programmer might not know automata and formal languages . . .)
- Problem 1: existence alone does not give an algorithm!
- Problem 2: automaton could be very big!

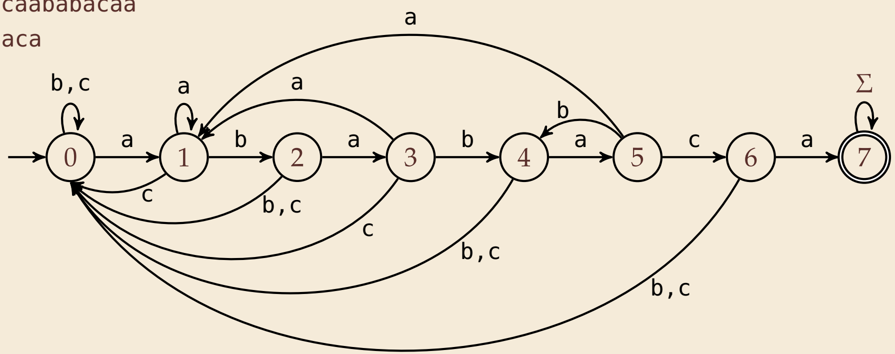
String matching with DFA

- Assume first, we already have a deterministic automaton
- How does string matching work?

Example:

T = aabacaababacaa

P = ababaca

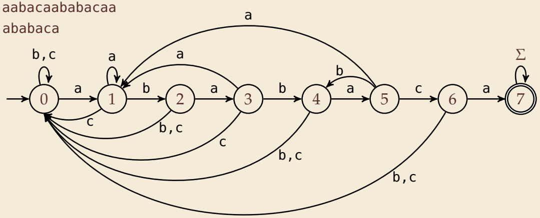


text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	1	1	2	3	0	1	1	2	3	4	5	6	7	7

String matching DFA – Intuition

Why does this work?

T = aabacaababacaa
P = ababaca



- Main insight:

  State q means:
  "we have seen $P[0..q)$ until here (but not any longer prefix of P)"

text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	1	1	2	3	0	1	1	2	3	4	5	6	7	7

- If the next text character c does not match, we know:
  - (i) text seen so far ends with $P[0\ldots q)\cdot c$
  - (ii) $P[0\ldots q)\cdot c$ is not a prefix of P
  - (iii) without reading c, $P[0..q)$ was the *longest* prefix of P that ends here.

$T = \cdots \boxed{P[0..q)}\boxed{c}$
$\boxed{P[0..q')}$
with $q' < q$

$\rightsquigarrow$ New longest matched prefix will be (weakly) shorter than q

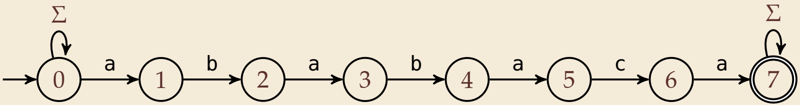
$\rightsquigarrow$ All information about the text needed to determine it is contained in $P[0\ldots q)\cdot c$!

6.4 Constructing String Matching Automata

NFA instead of DFA?

It remains to *construct* the DFA.

- trivial part:



- that actually is a ***non**deterministic finite automaton* (NFA) for $\Sigma^\star P\,\Sigma^\star$

⇝ We *could* use the NFA directly for string matching:

  - at any point in time, we are in a ***set* of states**
  - accept when one of them is final state

Example:

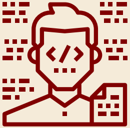
text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	0, 1	0, 1	0, 2	0, 1, 3	0	0, 1	0, 1	0, 2	0, 1, 3	0, 2, 4	0, 1, 3, 5	0, 6	0, 1, 7	0, 1, 7

But maintaining a whole set makes this slow . . .

Computing DFA directly

You have an NFA and want a DFA?
Simply apply the power-set construction
(and maybe DFA minimization)!

```
The powerset method has exponential state blow up!
        I guess I might as well use brute force ...
```



Ingenious algorithm by Knuth, Morris, and Pratt: construct DFA *inductively*:

Suppose we add character $P[j]$ to automaton A_j for $P[0..j)$ to construct A_{j+1}

- add new state and matching transition $\rightsquigarrow$ easy
- for each $c \neq P[j]$, we need $\delta(j, c)$ (transition from j when reading c)
- $\delta(j, c)$ = length of the longest prefix of $P[0..j)c$ that is a suffix of $P[1..j)c$
  = state of automaton after reading $P[1..j)c$
  $\leq$ j $\rightsquigarrow$ can use known automaton A_j for that!

State q means:
"we have seen $P[0..q)$ until here (but not any longer prefix of P)"

$\rightsquigarrow$ can directly compute A_{j+1} from A_j!

seems to require simulating automata $m \cdot \sigma$ times

Computing DFA efficiently

- **KMP's second insight:** simulations in one step differ only in last symbol

$\rightsquigarrow$ simply maintain state x, the state after reading $P[1..j)$.

  - copy its transitions
  - update x by following transitions for $P[j]$

```
procedure constructDFA(P[0..m)):
    // δ[q][c] = target state when reading c in state q
    for c ∈ Σ do
        δ[0][c] := 0
    δ[0][P[0]] := 1
    x := 0
    for j = 1, ..., m − 1 do
        for c ∈ Σ do // copy transitions
            δ[j][c] := δ[x][c]
        δ[j][P[j]] := j + 1 // match edge
        x := δ[x][P[j]] // update x
```

Example: $P[0..6) =$ ababac

$\delta(c, q)$	0	1	2	3	4	5
a	1	1	3	1	5	1
b	0	2	0	4	0	4
c	0	0	0	0	0	6

String matching with DFA – Discussion

- **Time:**
  - Matching: n table lookups for DFA transitions
  - building DFA: $\Theta(m\sigma)$ time (constant time per transition edge).
  - $\rightsquigarrow$ $\Theta(m\sigma + n)$ time for string matching.

- **Space:**
  - $\Theta(m\sigma)$ space for transition matrix.

👍 **fast matching** time   actually: hard to beat!

👍 total time asymptotically optimal for small alphabet   (for $\sigma = O(n/m)$)

👎 substantial **space overhead**, in particular for large alphabets

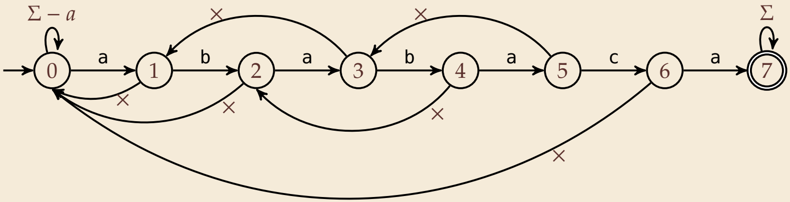
6.5 The Knuth-Morris-Pratt algorithm

Failure Links

- Recall: String matching with is DFA fast,
  but needs table of $m \times \sigma$ transitions.
- in fast DFA construction, we used that all simulations differ only by *last* symbol

$\rightsquigarrow$ **KMP's third insight:** do this last step of simulation from state x during *matching*!
. . . but how?

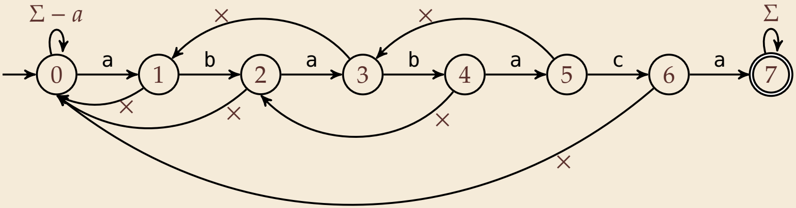
- **Answer:** Use a new type of transition: $\times$, the *failure links*
  - Use this transition (only) if no other one fits.
  - $\times$ *does not consume a character.* $\rightsquigarrow$ might follow several failure links



$\rightsquigarrow$ Computations are deterministic (but automaton is not a real DFA.)

Failure link automaton – Example

Example: T = `abababaaaca`, P = `ababaca`



T :	a	b	a	b	a	b	a	a	b	a	b	
P :	a	b	a	b	a	×						to state 3
			(a)	(b)	(a)	b	a	×				to state 1
								a	b	a	b	
q :	1	2	3	4	5	3, 4	5	3, 1, 0, 1	2	3	4	

(after reading this character)

The Knuth-Morris-Pratt Algorithm

```
procedure KMP(T[0..n), P[0..m)):
    fail[0..m] := failureLinks(P)
    i := 0 // current position in T
    q := 0 // current state of KMP automaton
    while i < n do
        if T[i] == P[q] then
            i := i + 1;  q := q + 1
            if q == m then
                return i − q // occurrence found
        else // i.e. T[i] ≠ P[q]
            if q ≥ 1 then
                q := fail[q] // follow one ×
            else
                i := i + 1
    end while
    return NO_MATCH
```

- only need single array *fail* for failure links
- (procedure failureLinks later)

Analysis: (matching part)

- always have $fail[j] < j$ for $j \geq 1$

$\rightsquigarrow$ in each iteration

  - either advance position in text ($i := i + 1$)
  - or shift pattern forward (guess $i - q$)

- each can happen at most n times

$\rightsquigarrow$ $\leq 2n$ symbol comparisons!

Computing failure links

- failure links point to error state x (from DFA construction)

$\rightsquigarrow$ run same algorithm, but store $\mathit{fail}[j] := x$ instead of copying all transitions

```
procedure failureLinks(P[0..m)):
    fail[0] := 0
    x := 0
    for j := 1, ..., m − 1 do
        fail[j] := x
        // update failure state using failure links:
        while P[x] ≠ P[j]
            if x == 0 then
                x := −1;  break
            else
                x := fail[x]
        end while
        x := x + 1
    end for
```

Analysis:

- m iterations of for loop
- while loop always decrements x
- x is incremented only once per iteration of for loop

$\rightsquigarrow$ $\leq m$ iterations of while loop *in total*

$\rightsquigarrow$ $\leq 2m$ symbol comparisons

Knuth-Morris-Pratt – Discussion

- **Time:**
  - $\leq 2n + 2m = O(n + m)$ character comparisons
  - clearly must at least *read* both T and P
  - $\rightsquigarrow$ KMP has optimal worst-case complexity!

- **Space:**
  - $\Theta(m)$ space for failure links

👍 total time asymptotically optimal (for any alphabet size)

👍 reasonable extra space

The KMP prefix function

- It turns out that the failure links are useful beyond KMP
- a slight variation is (more?) widely used: (for historic reasons)
  the (KMP) *prefix function* $F : [1..m-1] \to [0..m-1]$:

  $F[j]$ is the length of the longest prefix of $P[0..j]$ that is a suffix of $P[1..j]$.

- Can show: $\mathit{fail}[j] = F[j-1]$ for $j \geq 1$, and hence

  $\mathit{fail}[q]$ = ***length*** *of the longest prefix of $P[0..q)$ that is a suffix of $P[1..q)$.*

  memorize this!

- EAA Buch: String indices are 1-based, but definition of failure links matches! $\boxed{\Pi_P(q) = \mathit{fail}[q]}$
  $\Pi_P : [1..m] \to [0..m-1]$ with $\Pi_P(q) = \max\{k \in \mathbb{N}_0 : k < q \wedge P[0..k) \sqsupset P[0..q)]\} = \mathit{fail}[q]$

6.6 Beyond Optimal? The Boyer-Moore Algorithm

Motivation

- KMP is an optimal algorithm, isn't it? What else could we hope for?
- KMP is "only" optimal in the worst-case (and up to constant factors)
- how many comparisons do we need for the following instance?
  T = aaaaaaaaaaaaaaaa, P = xxxxx
  - there are no matches
  - we can *certify* the correctness of that output with only *4* comparisons:

T	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a
					x											
										x						
															x	
																x

⇝ **We did *not* even read most characters!**

Boyer-Moore Algorithm

- Let's check guesses *from right to left*!
- If we are lucky, we can eliminate several shifts in one shot!

⚠ must avoid (excessive) redundant checks, e. g., for $T = a^n$, $P = ba^{m-1}$

⇝ New rules:

- **Bad character jumps**: Upon mismatch at $T[i] = c$:
  - If P does not contain c, shift P entirely past i!
  - Otherwise, shift P to align the *last occurrence* of c in P with $T[i]$.
- **Good suffix jumps**:
  Upon a mismatch, shift so that the already matched *suffix* of P aligns with a previous occurrence of that suffix (or part of it) in P.
  (Details follow; ideas similar to KMP failure links)

⇝ two possible shifts (next guesses); use larger jump.

Boyer-Moore Algorithm – Code

```
procedure boyerMoore(T[0..n), P[0..m)):
    λ := computeLastOccurrences(P)
    γ := computeGoodSuffixes(P)
    i := 0 // current guess
    while i ≤ n − m
        j := m − 1 // next position in P to check
        while j ≥ 0 ∧ P[j] == T[i + j] do
            j := j − 1
        if j == −1 then
            return i
        else
            i := i + max{j − λ[T[i + j]], γ[j]}
    return NO_MATCH
```

- λ and γ explained below
- shift forward is larger of two heuristics
- shift is always positive (see below)

Bad character examples

P = `a l d o`

T = `w h e r e i s w a l d o`

w	h	e	r	e	i	s	w	a	l	d	o
			o								
							o				
								a	l	d	o

⇝ 6 characters not looked at

P = `m o o r e`

T = `b o y e r m o o r e`

b	o	y	e	r	m	o	o	r	e
				e					
				(r)	e				
					(m)	o	o	r	e

⇝ 4 characters not looked at

Last-Occurrence Function

- Preprocess pattern P and alphabet Σ
- *last-occurrence function* $\lambda[c]$ defined as
  - the largest index i such that $P[i] = c$ or
  - -1 if no such index exists
- **Example:** $P =$ `moore`

c	`m`	`o`	`r`	`e`	all others
$\lambda[c]$	0	2	3	4	-1

$P =$ `m o o r e`

$T =$ `b o y e r m o o r e`

				e					
				(r)	e				

$i = 0,\ j = 4,\ T[i+j] = r,\ \lambda[r] = 3$

$\rightsquigarrow$ shift by $j - \lambda[T[i+j]] = 1$

- λ computed in $O(m + \sigma)$ time.
- store as array $\lambda[0..\sigma)$.

```
procedure computeLastOccurrences(P[0..m)):
    λ[0..σ) := array initialized to 0
    for j = 0, ..., m − 1
        λ[P[j]] := j
    return λ
```

Good suffix examples

1. P = `sells␣shells`

s	h	e	i	l	a	␣	s	e	l	l	s	␣	s	h	e	l	l	s
							h	e	l	l	s						
								(e)	(l)	(l)	(s)						

2. P = `odetofood`

i	l	i	k	e	f	o	o	d	f	r	o	m	m	e	x	i	c	o	
				o	f	o	o	d											
							(o)	(d)											

matched suffix

- **Crucial ingredient:** longest suffix of $P[j+1..m)$ that occurs earlier in P.
- 2 cases (as illustrated above)
  1. complete suffix occurs in P $\rightsquigarrow$ characters left of suffix are *not* known to match
  2. part of suffix occurs at beginning of P

Good suffix jumps

- Precompute *good suffix jumps* $\gamma[0..m)$:
  - For $0 \le j < m$, $\gamma[j]$ stores shift if search failed at $P[j]$
  - At this point, had $T[i{+}j{+}1 \mathinner{..} i{+}m) \;=\; P[j{+}1 \mathinner{..} m)$, but $T[i] \ne P[j]$

$\leadsto$ $\gamma[j]$ is the shift $m - \ell$ for the *largest* ℓ such that

- $P[j{+}1..m)$ is a suffix of $P[0..\ell)$ and $P[j] \ne P[j{-}(m-\ell)]$

							h	e	l	l	s							
							×	(e)	(l)	(l)	(s)							

–OR–

- $P[0..\ell)$ is a suffix of $P[j{+}1..m)$

				o	f	o	o	d										
							(o)	(d)										

- Computable (similar to KMP failure function) in $\Theta(m)$ time.

Good suffix jumps – Efficient Computation

```
procedure computeGoodSuffixes(P[0..m]):
    fail[0..m] := failureLinks(P)
    revFail[0..m] := failureLinks(reverseString(P))
    γ[0..m) := new array initilized to m − fail[m]
    for ℓ := 1, . . . , m
        j := m − revFail[ℓ] − 1
        if γ[j] > ℓ − fail[ℓ]
            γ[j] := ℓ − revFail[ℓ]
        end if
    end for
    return γ
```

- Reuses failureLinks function from KMP
  - on both P and the reversed pattern!
- Correctness not obvious . . . Requires careful analysis of all possible cases
- Clearly $\Theta(m)$ time

Boyer-Moore algorithm – Discussion

- 👍 Worst-case running time $\in O(n + m + \sigma)$ **if** P does *not* occur in T.
  (follows from not at all obvious analysis!)

- 👎 As given, worst-case running time $\Theta(nm)$ if we want to report all occurrences
  - To avoid that, have to keep track of implied matches.
    (tricky because they can be in the "middle" of P)
  - Note: KMP reports all matches in $O(n + m)$ without modifications!

- 👍 On typical *English text*, Boyer Moore probes only approx. 25% of the characters in T!
  - $\rightsquigarrow$ Faster than KMP on English text.

- 👍 requires moderate extra space $\Theta(m + \sigma)$

6.7 The Rabin-Karp Algorithm

Space – The final frontier

- Knuth-Morris-Pratt has great worst case and real-time guarantees
- Boyer-Moore has great typical behavior
- What else to hope for?

- All require $\Omega(m)$ extra space;
  can be substantial for large patterns!
- Can we avoid that?

Rabin-Karp Fingerprint Algorithm – Idea

Idea: use *hashing* (but without explicit hash tables)

- Precompute & store only *hash* of pattern
- Compute hash for each guess
- If hashes agree, check characterwise

Example: (treat (sub)strings as decimal numbers)

$P = 59265$

$T = 31415926535897932238$

Hash function: $h(x) = x \bmod 97$

$\rightsquigarrow \quad h(P) = 95.$

3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8

$h(31415) = 84$

$h(14159) = 94$

$h(41592) = 76$

$h(15926) = 18$

$\mathbf{h(59262) = 95}$

Rabin-Karp Fingerprint Algorithm – First Attempt

```
procedure rabinKarpSimplistic(T[0..n), P[0..m)):
    M := suitable prime number
    h_P := computeHash(P[0..m), M)
    for i := 0, ..., n − m do
        h_T := computeHash(T[i..i + m), M)
        if h_T == h_P then
            if T[i..i + m) == P // m comparisons
                then return i
    return NO_MATCH
```

- never misses a match since $h(S_1) \neq h(S_2)$ implies $S_1 \neq S_2$
- $h(T[k..k+m)$ depends on m characters $\rightsquigarrow$ naive computation takes $\Theta(m)$ time

$\rightsquigarrow$ Running time is $\Theta(mn)$ for search miss ... can we improve this?

Rabin-Karp Fingerprint Algorithm – Fast Rehash

- **Crucial insight:** We can update hashes in constant time.
  - Use previous hash to compute next hash
  - $O(1)$ time per hash, except first one

for above hash function!

Example:

- Pre-compute: $10000 \bmod 97 = 9$
- Previous hash: $41592 \bmod 97 = 76$
- Next hash: $15926 \bmod 97 = ??$

Observation:

$$
\begin{aligned}
15926 \bmod 97 &= (41592 - (4 \cdot 10000)) \cdot 10 + 6 \quad \bmod 97 \\
&= (76 - (4 \cdot 9)) \cdot 10 + 6 \quad \bmod 97 \\
&= 406 \bmod 97 = 18
\end{aligned}
$$

Rabin-Karp Fingerprint Algorithm – Code

- use a convenient radix $R \geq \sigma$ ($R = 10$ in our examples; $R = 2^k$ is faster)
- Choose modulus M at *random* to be huge prime (randomization against worst-case inputs)
  $\rightsquigarrow$ false positive probability $\approx 1/M$
- all numbers remain $\leq 2R^2$ $\rightsquigarrow$ $O(1)$ time arithmetic on word-RAM

```
procedure rabinKarp(T[0..n), P[0..m), R):
    M := suitable prime number
    h_P := computeHash(P[0..m), M)
    h_T := computeHash(T[0..m), M)
    s := R^(m−1) mod M
    for i := 0, ..., n − m do
        if h_T == h_P then
            if T[i..i + m) = P
                return i
        if i < n − m then
            h_T := ((h_T − T[i] · s) · R + T[i + m]) mod M
    return NO_MATCH
```

Rabin-Karp – Discussion

- 👍 Expected running time is $O(m + n)$
- 👎 $\Theta(mn)$ worst-case;
  but this is very unlikely
- 👍 Extends to 2D patterns and other generalizations
- 👍 Only constant extra space!

String Matching Conclusion

	Brute-Force	DFA	KMP	BM	RK	Suffix trees*
Preproc. time	—	$O(m\sigma)$	$O(m)$	$O(m+\sigma)$	$O(m)$	$O(n)$
Search time	$O(nm)$	$O(n)$	$O(n)$	$O(n)$ (often better)	$O(n+m)$ (expected)	$O(m)$
Extra space	—	$O(m\sigma)$	$O(m)$	$O(m+\sigma)$	$O(1)$	$O(n)$

* (see Unit 13)