Strings: Substring Search

Neerja Mhaskar

Dept. of Computing and Software, McMaster University, Canada

Acknowledgments: Material partially based on the textbook Algorithms by Robert Sedgewick and Kevin Wayne (Chapters 5.3)

Some string definitions

A **string** is a finite array $w[1..n]$ of elements chosen from a set of totally ordered symbols Σ, called an **alphabet**. We write $\sigma = |\Sigma|$

Example: $w = abaababaabaaab$ on $\Sigma = \{a, b\}$.

The **length** of the string is written $|w|$. If $|w| = 0$, then the string w is **empty** and written as ε. Typically $|w| >> \sigma$.

If $w = uxv$ for strings u, x, v, then u is called a **prefix**, x is called a **substring** or **factor**, and v is called a **suffix** of w.

Example: $w = ababaabb$, $abab$ is a prefix, $abaa$ is a substring, and $aabb$ is a suffix.

A substring u of w is **proper** if $|u| < |w|$.

Substring Search (Pattern matching) Problem

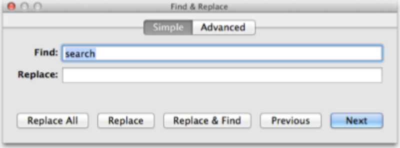
Goal. Find pattern of length M in a text of length N.

typically N >> M

pattern → N E E D L E

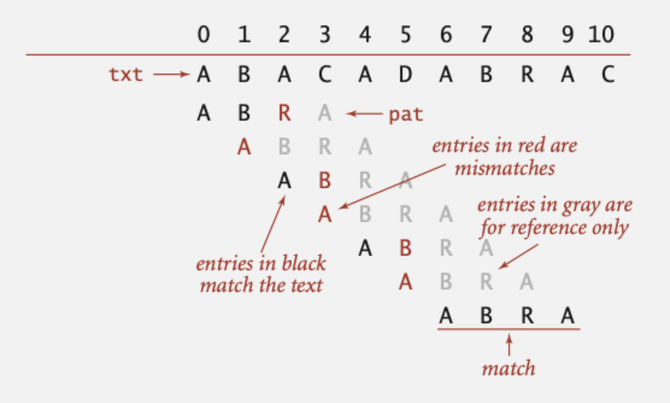
text → I N A H A Y S T A C K N E E D L E I N A

match



Patching Matching: Brute Force

Align the pattern at each index position of the text.



Patching Matching Brute Force: JAVA

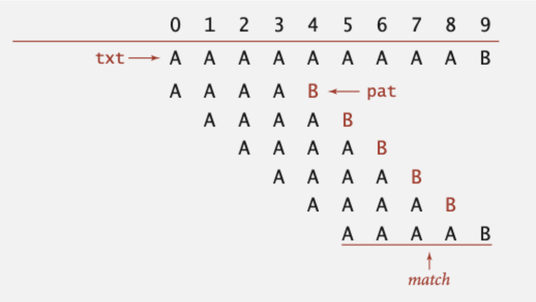
Align the pattern at each index position of the text.

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

```
public static int search(String pat, String txt)
{
   int M = pat.length();
   int N = txt.length();
   for (int i = 0; i <= N - M; i++)
   {
      int j;
      for (j = 0; j < M; j++)
         if (txt.charAt(i+j) != pat.charAt(j))
            break;
      if (j == M) return i;        index in text where
                                   pattern starts
   }
   return N;        not found
}
```

Brute-force substring search: worst case

Brute-force algorithm can be slow if text and pattern are repetitive.



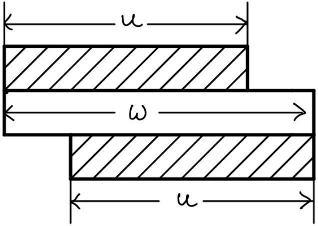
Worst case. ~MN character compares.

The Knuth-Morris-Pratt (KMP) Algorithm

- The most famous pattern-matching algorithm.
- Preprocessing: compute the longest border of every prefix of p [stay tuned].
- Runs in O(N+M) time, where N is the length of the text and M is the length of the pattern.
- However, not very fast in practice.

Borders

A string $\boldsymbol{u}$ is said to be a **border** of a string $\boldsymbol{w}$ if it is both a prefix and a suffix of w, and of length $< |\boldsymbol{w}|$.



For example: $\boldsymbol{u} = aba$ is a border of $\boldsymbol{w} = abacaba$. In fact w has three borders ε, a, aba.

A string $\boldsymbol{w}$ can have at most $|\boldsymbol{w}|$ borders (including the empty border $= \varepsilon$).

For example $\boldsymbol{w}[1..5] = aaaaa$, has borders $\varepsilon, a, aa, aaa, aaaa$.

Border Array

A **border array** $\boldsymbol{\beta_x}$ of $\boldsymbol{x}$, is an integer array of length n, where the i-th element of the array is equal the length of the longest border of $\boldsymbol{x}[1..i]$.

For example: The border array $\beta_{\boldsymbol{w}}$ of $\boldsymbol{w} = abacaba$ is shown below:

	1	2	3	4	5	6	7
$\boldsymbol{w}$	a	b	a	c	a	b	a
$\beta_{\boldsymbol{w}}$	0	0	1	0	1	2	3

$\beta[1] = 0 =$ the length of the longest border (ε) of $\boldsymbol{w}[1..1] = a$.

$\beta[2] = 0 =$ the length of the longest border (ε) of $\boldsymbol{w}[1..2] = ab$.

$\beta[3] = 1 =$ the length of the longest border (a) of $\boldsymbol{w}[1..3] = aba$.

$\beta[4] = 0 =$ the length of the longest border (ε) of $\boldsymbol{w}[1..4] = abac$.

$\beta[5] = 1 =$ the length of the longest border (a) of $\boldsymbol{w}[1..5] = abaca$.

$\beta[6] = 2 =$ the length of the longest border (ab) of $\boldsymbol{w}[1..6] = abacab$.

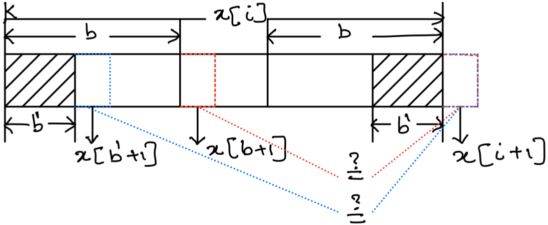
$\beta[7] = 3 =$ the length of the longest border (aba) of $\boldsymbol{w}[1..7] = abacaba$.

Algorithm for computing the Border Array

procedure COMPUTE_BORDER ARRRAY
  $\beta[1] = 0$
  for $i = 1$ to $n - 1$ **do**
    $b = \beta[i]$
    while $b > 0$ and $\boldsymbol{x}[i+1] \neq \boldsymbol{x}[b+1]$ **do**
      $b = \beta[b]$
    if $\boldsymbol{x}[i+1] = \boldsymbol{x}[b+1]$ **then**
      $\beta[i+1] = b + 1$
    else
      $\beta[i+1] = 0$

Compute Border Array: Explanation

- While computing the border array $\beta[i]$, the algorithm first checks if the current longest border $= \beta[i-1]$ can be extended. This is only possible when $\boldsymbol{x}[i+1] = \boldsymbol{x}[b+1]$.
- If the above is not satisfied, it checks for the second longest border $= \beta[b]$, can be extended. This is only possible when $\boldsymbol{x}[i+1] = \boldsymbol{x}[b+1]$.
- The algorithm continues to check the third longest border, the 4th longest border,... and so on till either the border can be extended or $b = 0$.

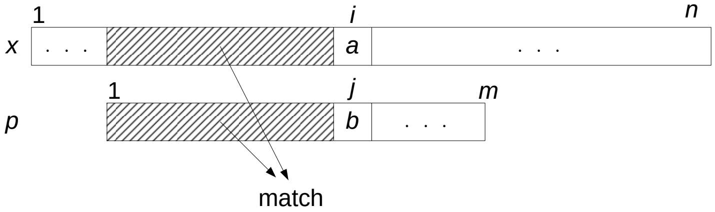


Compute Border Array: Analysis

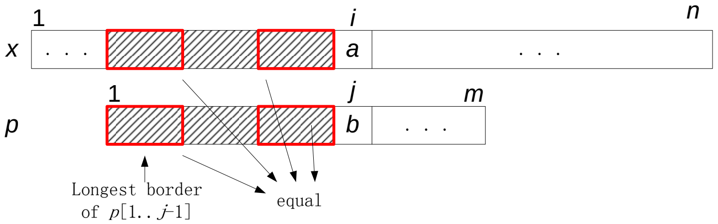
Proposition. The Compute_Border Array procedure correctly computes the border array.

Proposition. The Compute_Border Array procedure requires $\Theta(n)$ time and constant additional space.

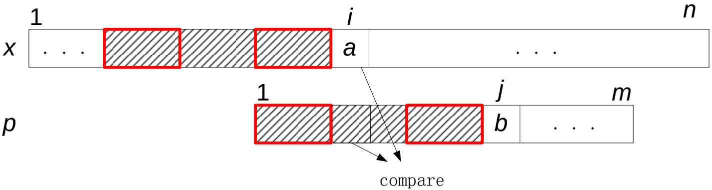
The KMP Algorithm Intuition - I



The KMP Algorithm Intuition - 2



The KMP Algorithm Intuition - 3



KMP Algorithm

procedure $\mathrm{KMP}(\boldsymbol{w}, \boldsymbol{p})$
    $i = 0$ $\triangleright$ $i =$ the index position in $\boldsymbol{w}$ such that $\boldsymbol{w}[1..i]$ is searched.
    $j = 0$ $\triangleright$ $j =$ the index position in $\boldsymbol{p}$ such that $\boldsymbol{p}[1..j]$ is matched.
    $indexlist = \emptyset$ $\triangleright$ List of indices where $\boldsymbol{p}$ occurs in $\boldsymbol{w}$
    $\beta_{\boldsymbol{p}} \leftarrow$ `Border array of pattern` $\boldsymbol{p}$
    while $i < n$ **do**
        if $\boldsymbol{p}[j+1] = \boldsymbol{w}[i+1]$ **then**
            $j = j + 1; i = i + 1$
            if $j = m$ **then**
                $indexlist = indexlist \cup \{i - j + 1\}$
                $j = \beta_{\boldsymbol{p}}[j]$
        else
            if $j = 0$ **then**
                $i = i + 1$
            else
                $j = \beta_{\boldsymbol{p}}[j]$
    return $indexlist$

KMP Algorithm Explanation

- During the execution of the KMP algorithm, each time there is a match, we increment the current indices.
- On the other hand, if there is a mismatch and we have previously made progress in P (pattern), then we consult the border array of P to determine the new index in P where we need to continue checking P against T .
- Otherwise, if the length of the longest border of the prefix of P that matched with a substring of X is 0, then we are at the beginning of P, and so we check the next character in T with the first character in P.
- We repeat this process until we find a match of P in T or the index for T reaches n, the length of T (indicating that we did not find the pattern P in T).

KMP Algorithm Example

Text $= \boldsymbol{w} = abacababacabacaba$

Pattern $= \boldsymbol{p} = abacaba$, and $\beta_{\boldsymbol{p}} = 0010123$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
w	a	b	a	c	a	b	a	b	a	c	a	b	a	c	a	b	a
p	a	b	a	c	a	b	a										
					a	b	a	c									
							a	b	a	c	a	b	a				
											a	b	a	c	a	b	a

Pattern is first aligned at $i = 1$. Both i, j are incremented till $i, j = 7$, since $j = m = 7$, pattern occurs at $x[1]$ and so $indexlist = \{1\}$. Pattern is then aligned at $i = 8 - 3 = 5$, as $j = \beta_{\boldsymbol{p}}[7] = 3$. Then, we check if $\boldsymbol{x}[8]$, $\boldsymbol{p}[4]$ match, and so on. That is, we continue searching for other occurrences of the pattern in a similar manner. Therefore, at the end we get $indexlist = \{1, 7, 11\}$.

KMP Algorithm: Analysis

The KMP algorithm runs in O(N+M) time.

Boyer Moore Algorithm Outline

- Scan text from left to right (just like KMP).
- But, scan characters in pattern (for matches) from right to left.
- Perform shift = max.shift by applying the bad character rule.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
text →	F	I	N	D	I	N	A	H	A	Y	S	T	A	C	K	N	E	E	D	L	E	I	N	A
	N	E	E	D	L	E ← *pattern*																		
						N	E	E	D	L	E													
												N	E	E	D	L	E							
																N	E	E	D	L	E			

Boyer Moore Algorithm: Bad Character Rule

Bad Character Rule: Upon mismatch, skip alignments until:

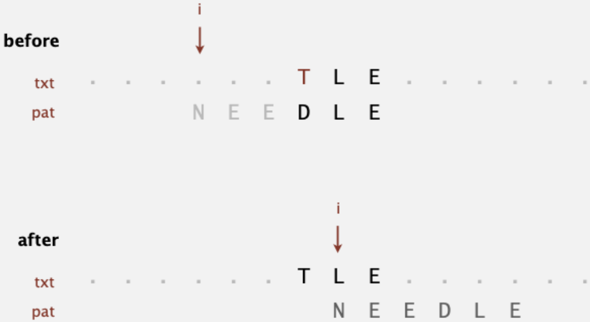
(a) mismatch becomes a match, or

(b) the pattern moves past the mismatched character (bad character).

Can skip as many as M text chars when finding one not in the pattern.

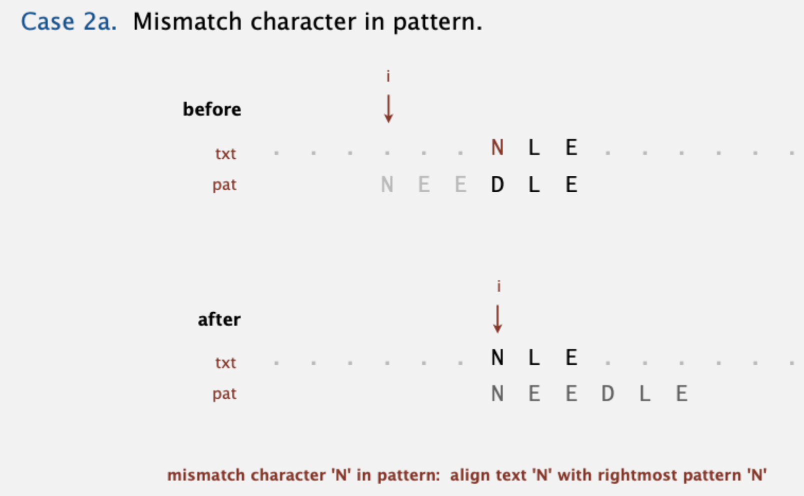
Boyer Moore Algorithm: Bad Character Rule Example I

Case 1. Mismatch character not in pattern.



mismatch character 'T' not in pattern: increment i one character beyond 'T'

Boyer Moore Algorithm: Bad Character Rule Example II



Boyer Moore Algorithm: Bad Character Rule Example III

Q. How much to skip?

Case 2b. Mismatch character in pattern (but heuristic no help).

i

before

txt . . . . . . E L E . . . . . .

pat N E E D L E

i

after

txt . . . . . . E L E . . . . . .

pat N E E D L E

Boyer Moore Algorithm: Bad Character Rule

Q. How to compute the shift?

A. Precompute index of rightmost occurrence of character c in a given prefix of the pattern. (-1 if character not in pattern). Maintain this information in a table (skip table).

$skiptable[i, j] =$ the index of rightmost occurrence of character i in prefix of length $j - 1$ in the pattern.

Pattern of length M

			0	1	2	3	4	5
			N	**E**	**E**	**D**	**L**	**E**
	0-2	**A-C**	-1	-1	-1	-1	-1	-1
	3	**D**	-1	-1	-1	-1	3	3
Σ	4	**E**	-1	-1	1	2	2	2
	5-10	**F-K**	-1	-1	-1	-1	-1	-1
	11	**L**	-1	-1	-1	-1	-1	4
	12	**M**	-1	-1	-1	-1	-1	-1
	13	**N**	-1	0	0	0	0	0
	14-25	**O-Z**	-1	-1	-1	-1	-1	-1

Skiptable

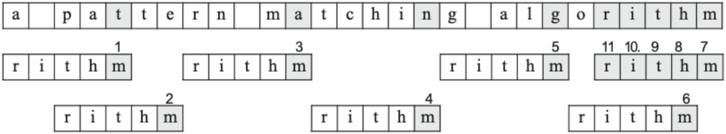
Boyer Moore Algorithm: Java Implementation I

```
public int search(String txt)
{
   int N = txt.length();
   int M = pat.length();
   int skip;
   for (int i = 0; i <= N-M; i += skip)
   {
      skip = 0;
      for (int j = M-1; j >= 0; j--)
      {
         if (pat.charAt(j) != txt.charAt(i+j))
         {
            skip = Max(1, j-skiptable[txt.charAt(i+i), j])
            break;
         }
      }
      if (skip == 0) return i;
   }
   return N;
}
```

Boyer Moore Algorithm: Java Implementation II

- The algorithm returns the first index position at which the pattern is found in the text (when $skip = 0$)
- If the pattern is not found in the text it returns N. Note that the strings (text and pattern) are indexed from $0..N-1$
- $skip =$ no.of positions to skip (including the position of the current alignment) in the text.

Boyer Moore Algorithm: Example with Significant Speed Up



- Execution of the Boyer-Moore algorithm on an English text and pattern, where a significant speedup is achieved. Note that not all text characters are examined.
- For bigger alphabets and structured texts (like texts in English, etc.), Boyer-Moore is usually faster than Knuth-Morris-Pratt.

Boyer Moore Algorithm: Analysis

Property. Pattern matching/substring search with the Boyer-Moore's bad character rule takes at-least (and mostly) $\sim N/M$ character compares to search for a pattern of length M in a text of length N.

Worst-case. Can be as bad as MN.

i	skip		0	1	2	3	4	5	6	7	8	9
		txt →	B	B	B	B	B	B	B	B	B	B
0	0		A	B	B	B	B	← pat				
1	1			A	B	B	B	B				
2	1				A	B	B	B	B			
3	1					A	B	B	B	B		
4	1						A	B	B	B	B	
5	1							A	B	B	B	B

Boyer-Moore variant. Can improve worst case to $\sim 3N$ character compares by adding a KMP-like rule to guard against repetitive patterns.

Rabin Karp

Basic idea Use hashing for pattern matching.

- Compute a hash of pat[0..M-1].
- For each $1 \leq i \leq n$, compute a hash of txt[i..M+i-1].
    - If hash of pat[0..M-1] $=$ txt[i..M+i-1] hash, then align pattern at index i, and
    - Perform a brute force comparison for check for a match.

Rabin Karp: Example

pat.charAt(i)

i	0	1	2	3	4	
	2	6	5	3	5	% 997 = 613

txt.charAt(i)

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3
0	3	1	4	1	5	% 997 = 508										
1		1	4	1	5	9	% 997 = 201									
2			4	1	5	9	2	% 997 = 715								
3				1	5	9	2	6	% 997 = 971							
4					5	9	2	6	5	% 997 = 442						
5						9	2	6	5	3	% 997 = 929				*match*	
6 ← *return* i = 6							2	6	5	3	5	% 997 = 613				

Rabin Karp: Modular hashing

- Modular hashing good for Rabin Karp as it works on long keys such as strings of length N. Why?

  Answer:
  - Treat strings as concatenation of numbers
  - Then take a mod of this number to compute the string's hash.
- If R is greater than any character value we can treat the string as an N-digit base-R integer.
- Then to compute the strings hash, we simply compute the remainder that results when dividing that number by Q; that is, we use the hash function to be $h(x) = x \mod Q$.
- For instance, in the previous example, $R = 10$ and $R > c, \texttt{ where } c \in \{0, 1, 2, \ldots, 9\}$. We treat the pattern 2 6 5 3 5 as a 5 digit base 10 integer.
- Then the hash of the pattern is $26535 \mod 997 = 613$.

Modular Arithmetic and Horner's method I

Treating strings as concatenation of numbers – results in huge integers. How to address this?

Solution: Use Modular Arithmetic and Horner's method!

To keep numbers small, take intermediate results modulo Q.

1. $(a+b) \mod Q = ((a \mod Q) + (b \mod Q)) \mod Q$
2. $(a*b) \mod Q = ((a \mod Q) * (b \mod Q)) \mod Q$
3. $-a \mod Q = Q - a$
4. $a \mod Q = a \underbrace{\mod Q \mod Q \ldots \mod Q}_{N}$

Modular Arithmetic and Horner's method II

A classic algorithm known as Horner's method (based on Horner's rule (see below) for optimal polynomial evaluation) computes the polynomial with N multiplications, and additions operations.

$$\begin{aligned} & a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n \\ & \quad = a_0 + x\Big(a_1 + x\Big(a_2 + x\big(a_3 + \cdots + x(a_{n-1} + x\, a_n)\cdots\big)\Big)\Big). \end{aligned}$$

How to choose Q?

Since we are not actually building/storing a hash table, and just testing for a collision with one key, our pattern, we can choose Q to be:

- a prime number not close to a power of 2 and as large as you wish!
- The book choses a prime Q $> 10^{20}$, so that the probability that a random key hashes to the same value as our pattern is less than $1/10^{20} = 10^{-20}$.

Efficiently computing the hash function - I

- We use the notation t_i for `txt.charAt(i)`.
- Let $x_i = txt[i..M+i-1]$ and $h(x_i)$ be hash value of x_i . Then using the modular hash function, we get

$$\begin{aligned} h(x_i) &= h(t_i t_{i+1} \ldots t_{i+M-1}) \\ &= (t_i t_{i+1} \ldots t_{i+M-1}) \mod Q \\ &= (t_i R^{M-1} + t_{i+1} R^{M-2} + \ldots + t_{i+M-1} R^0) \mod Q \end{aligned} \tag{1}$$

Consider the example on slide 28 (R=10)

$$\begin{aligned} h(x_0) &= (t_0 t_1 \ldots t_4) \mod 997 \\ &= 31415 \mod 997 \\ &= (3 * 10^4 + 1 * 10^3 + 4 * 10^2 + 1 * 10^1 + 5 * 10^0) \mod 997 \\ &= 508 \end{aligned}$$

Efficiently computing the hash function - II

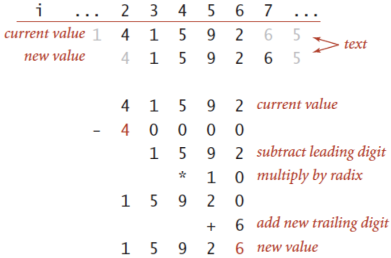
Challenge. How to efficiently compute x_{i+1} from x_i in constant time?
Answer. Use Rolling hash!

$$\text{Substring starting at i, } x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \ldots + t_{i+M-1} R^0$$

$$\text{Substring starting at i+1, } x_{i+1} = t_{i+1} R^{M-1} + t_{i+2} R^{M-2} + \ldots + t_{i+M} R^0,$$

$$\texttt{But,} \quad x_{i+1} = (x_i - t_i R^{M-1}) R + t_{i+M}$$

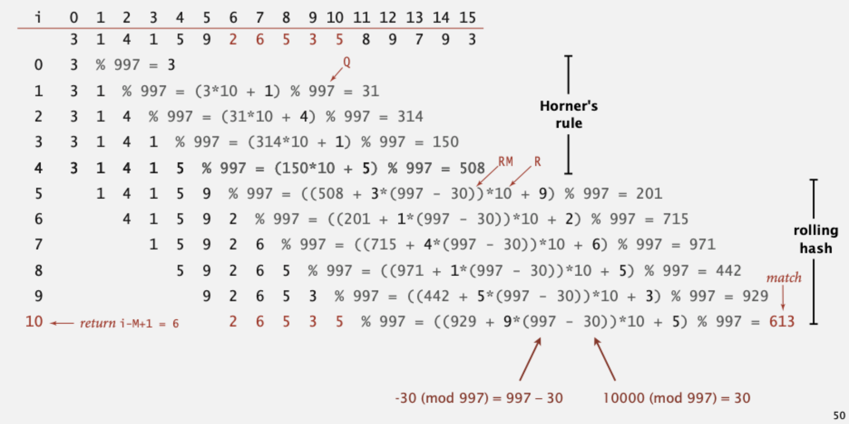
$$\texttt{Therefore,} \quad h(x_{i+1}) = ((x_i - t_i R^{M-1}) R + t_{i+M}) \mod Q$$



Rabin-Karp substring search example

First R entries: Use Horner's rule.

Remaining entries: Use rolling hash (and % to avoid overflow).



Rabin-Karp example Rolling hash explanation

Explanation of how the previous slide computes $h(x_{i+1})$ from $h(x_i)$

From the previous slides we know,

$$
\begin{aligned}
h(x_{i+1}) &= ((x_i - t_i R^{M-1})R + t_{i+M}) \mod Q \\
&= ((x_i - t_i R^{M-1})R \mod Q + t_{i+M} \mod Q) \mod Q \\
&= ((x_i \mod Q - t_i R^{M-1} \mod Q)R + t_{i+M} \mod Q) \mod Q \\
&= ((h(x_i) + t_i(-R^{M-1} \mod Q))R + t_{i+M}) \mod Q \\
&= ((h(x_i) + t_i(Q - (R^{M-1} \mod Q))R + t_{i+M}) \mod Q
\end{aligned}
$$

Consider the below snapshot of the previous slide.

```
4   3  1  4  1  5  % 997 = (150*10 + 5) % 997 = 508   RM   R
5      1  4  1  5  9  % 997 = ((508 + 3*(997 - 30))*10 + 9) % 997 = 201
```

To compute $h(14159)$, we use the above equation, and get

$h(14159) = ((508 + 3 * (997 - (10^4 \mod 997))) * 10 + 9)) \mod 997$

$h(14159) = ((508 + 3 * (997 - 30)) * 10 + 9)) \mod 997$

Rabin-Karp: Analysis

In the worst case, where $h(x_i) = h(x_{i+1})$ for all $1 \leq i \leq N - 1$, the Rabin-Karp algorithm runs in O(MN) time (same as brute force).

Rabin-Karp summary

- Rabin-Karp substring search is known as a fingerprint search because it uses a small amount of information to represent a (potentially very large) pattern.
- Then it looks for this fingerprint (the hash value) in the text.
- The algorithm is efficient because the fingerprints can be efficiently computed and compared.