Lecture #18
String Matching (1)

Algorithm

JBNU

Jinhong Jung

In This Lecture

- String matching
  - Problem definition

- Algorithms for string matching
  - Naïve algorithm
  - SAN (string-as-number) algorithm
  - Rabin-Karp algorithm

Outline

❑ String matching

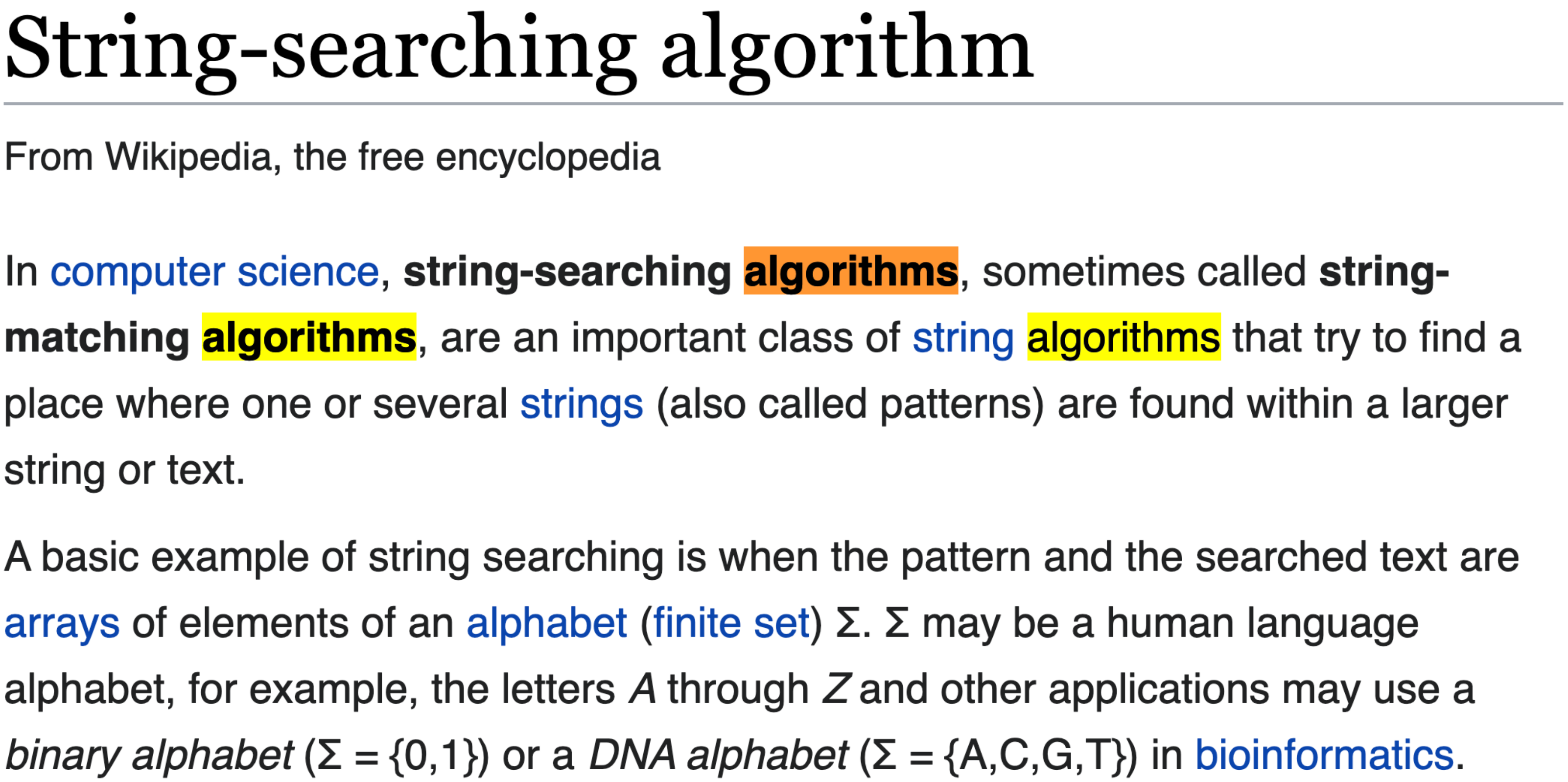
❑ Naïve algorithm

❑ SAN algorithm

❑ Rabin-Karp algorithm

String Matching (1)

- How can we efficiently find a word in a document?
  - Let's find "algorithms" in the following document.
  - The querying word is called pattern.



String-searching algorithm

From Wikipedia, the free encyclopedia

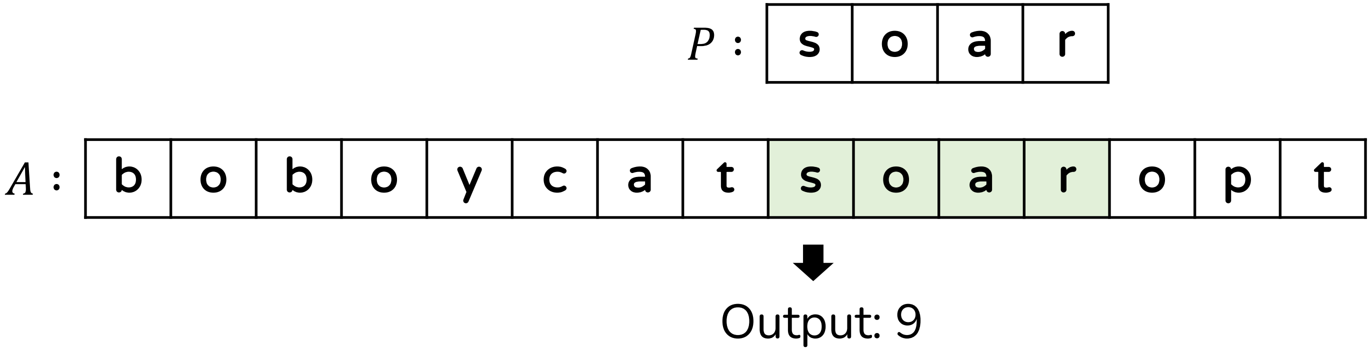
In computer science, **string-searching algorithms**, sometimes called **string-matching algorithms**, are an important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text.

A basic example of string searching is when the pattern and the searched text are arrays of elements of an alphabet (finite set) Σ. Σ may be a human language alphabet, for example, the letters *A* through *Z* and other applications may use a *binary alphabet* (Σ = {0,1}) or a *DNA alphabet* (Σ = {A,C,G,T}) in bioinformatics.

String Matching (2)

❑ Problem definition

- **Input**
  - Document string: $A[1 \cdots n]$ where n is # of characters of a document
  - Pattern string: $P[1 \cdots m]$ where m is # of characters of a pattern
    - In general, $m \ll n$
- **Output**
  - Locations where the pattern string is matched



P :	s	o	a	r

A :	b	o	b	o	y	c	a	t	s	o	a	r	o	p	t

Output: 9

Outline

❑ String matching

❑ Naïve algorithm

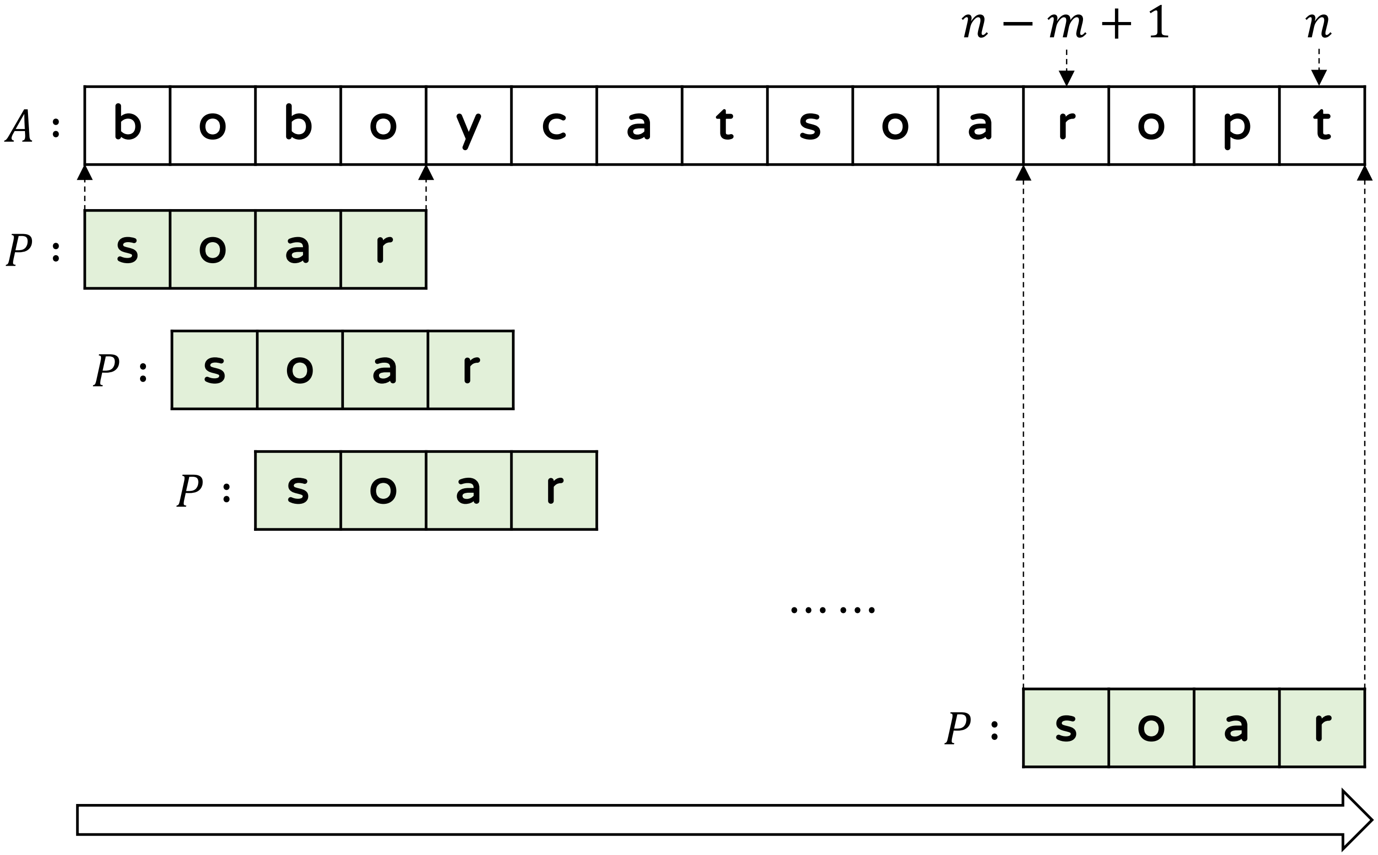
❑ SAN algorithm

❑ Rabin-Karp algorithm

Naïve Algorithm (1)

❑ Main idea

- Sequentially match the pattern with a sub-string of a document string from left to right



Naïve Algorithm (2)

Pseudocode

```
def naïve-matching(A, P):
    # n: length of A (document string)
    # m: length of P (pattern string)

    for i ← 1 to n-m+1:
        if P[1⋯m] == A[i ⋯i+m-1]
            output there is a matching at A[i]
```

- Time complexity is $O(mn)$
  - The for-loop of i repeats $O(n)$ time
  - For each step, the string comparison takes $O(m)$ time

How can we match more quickly than $O(mn)$?

Outline

❑ String matching

❑ Naïve algorithm

❑ SAN algorithm

❑ Rabin-Karp algorithm

String As Numbers (1)

❑ Assume that a string consists of decimal numbers

A:	1	0	3	4	5	3	1	6	1	0	1	2	3	7	5

P:	5	3	1	6	1

- Each string can be considered as a number
  - The pattern P is considered as 53161
  - The substring of A is considered as 45316

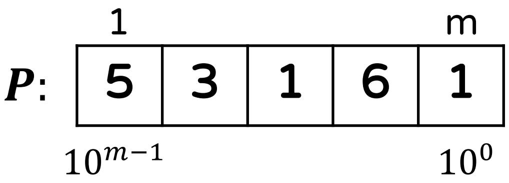
- If those numbers are known, they are compared in a constant time!

String As Numbers (2)

❑ How to convert a string to a number?

- Let $\boldsymbol{X}[i]$ be the i-th value (or character) of a string $\boldsymbol{X}$
- Let p be the number from the pattern $\boldsymbol{P}$

$$p = 10^0\boldsymbol{P}[m] + 10^1\boldsymbol{P}[m-1] + 10^2\boldsymbol{P}[m-2] + \cdots 10^{m-1}\boldsymbol{P}[1]$$

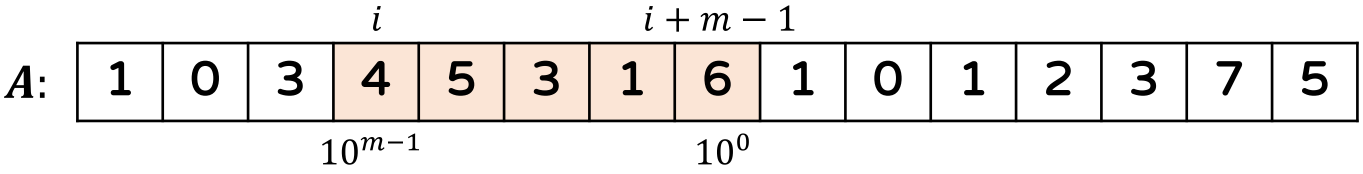


5	3	1	6	1

$$p = 10^0{\times}1 + 10^1{\times}6 + 10^2{\times}1 + 10^3{\times}3 + 10^4{\times}5$$

- Let a_i be the number from the substring $\boldsymbol{A}[i \cdots i+m-1]$

$$a_i = 10^0\boldsymbol{A}[i+m-1] + 10^1\boldsymbol{A}[i+m-2] + 10^2\boldsymbol{A}[i+m-3] + \cdots 10^{m-1}\boldsymbol{A}[i]$$



1	0	3	4	5	3	1	6	1	0	1	2	3	7	5

String As Numbers (3)

❑ How to quickly convert a string to a number?

- Let’s consider the length of a pattern is 4

$$p = 10^0\boldsymbol{P}[4] + 10^1\boldsymbol{P}[3] + 10^2\boldsymbol{P}[2] + 10^3\boldsymbol{P}[1]$$

- Then, we can group the right three terms as follows:

$$p = 10^0\boldsymbol{P}[4] + 10^1(\boldsymbol{P}[3] + 10^1\boldsymbol{P}[2] + 10^2\boldsymbol{P}[1])$$

- Repeat the above one more time

$$p = 10^0\boldsymbol{P}[4] + 10^1\big(\boldsymbol{P}[3] + 10^1(\boldsymbol{P}[2] + 10^1\boldsymbol{P}[1])\big)$$

- Initially, set p to 0; then, the final p is obtained as follows

$$p \leftarrow \boldsymbol{P}[1] + 10\times p$$
$$p \leftarrow \boldsymbol{P}[2] + 10\times p$$
$$p \leftarrow \boldsymbol{P}[3] + 10\times p$$
$$p \leftarrow \boldsymbol{P}[4] + 10\times p$$

```
p ← 0
for $i \leftarrow 1$ to 4 $(\Rightarrow m)$:
    $p \leftarrow \boldsymbol{P}[i] + 10\times p$
```

Converting a string of length m to a number takes $O(m)$ time

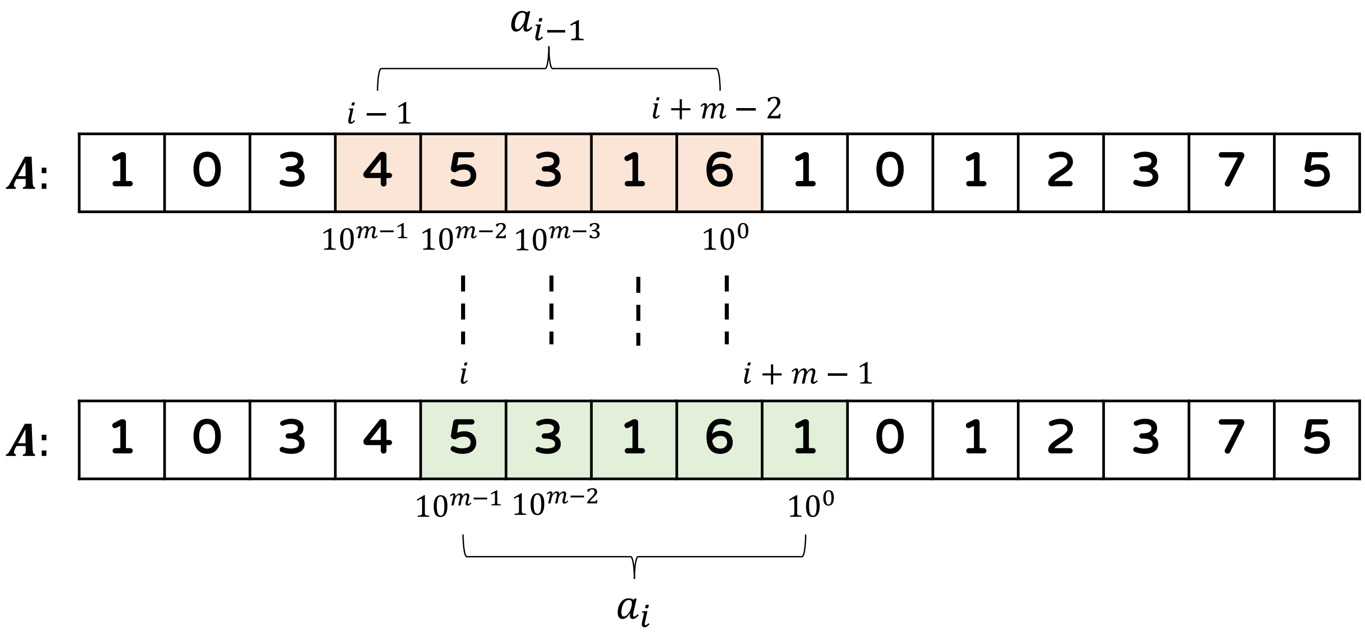
String As Numbers (4)

- Naïve approach for string-as-numbers
  - **Step 1)** Convert the pattern $\boldsymbol{P}$ to number p
  - For $i \leftarrow 1$ to $n - m + 1$
    - **Step 2)** Convert the document string $\boldsymbol{A}$'s substring at index i to a_i
    - **Step 3)** Check if p is the same as a_i
- Time complexity of this approach is also $O(mn)$
  - Because each Step 2 takes $O(m)$ time for $O(n)$ iterations
  - Same as the naïve algorithm

- Can we do this better?

String As Numbers (5)

- ❑ Number of a substring of $\boldsymbol{A}$ can be incrementally computed!



$$a_i = 10\times(a_{i-1} - 10^{m-1}\times\boldsymbol{A}[i-1]) + \boldsymbol{A}[i+m-1]$$

String As Numbers (6)

❑ Pseudocode of naïve string-as-numbers

- Under the assumption a string is in decimal numbers

n is the length of $\boldsymbol{A}$
m is the length of $\boldsymbol{P}$

```
def SAN-search(A, P):
    $p$ ← 0          # number of P
    $a_1$ ← 0        # number of sub-string of A at index 1

    for $i \leftarrow 1$ to $m$:
        $p \leftarrow \boldsymbol{P}[i] + 10 \times p$
        $a_1 \leftarrow \boldsymbol{A}[i] + 10 \times a_1$
                                                   $O(m)$ time

    for $i \leftarrow 1$ to $n - m + 1$:        Precompute this before
                                               the for-loop in $O(\log m)$
        if $i > 1$:
            $a_i \leftarrow 10 \times (a_{i-1} - \boxed{10^{m-1}} \times \boldsymbol{A}[i-1]) + \boldsymbol{A}[i+m-1]$
                                                   $O(n)$ time
        if $p$ == $a_i$:
            output "there is a matching at $\boldsymbol{A}[i]$"
```

- Time complexity is $O(m + n)$

String As Numbers (7)

❑ How to generalize SAN-search to a normal string?

- Suppose a string consists of unit characters in $\mathbf{\Sigma}$
  - For alphabets, $\mathbf{\Sigma} = \{a, b, c, \cdots, z\}$ and $|\mathbf{\Sigma}| = 26$
  - For ASCII codes, $|\mathbf{\Sigma}| = 128$
- Then, a string is considered as a number in base-$|\mathbf{\Sigma}|$ number system
- As before, we only consider numbers in base-10 number system
- By simply replacing 10 with $|\mathbf{\Sigma}|$, we can generalize SAN-search to a normal string

String As Numbers (8)

❑ Pseudocode of naïve string-as-numbers

n is the length of $\boldsymbol{A}$
m is the length of $\boldsymbol{P}$
d is $|\boldsymbol{\Sigma}|$

```
def SAN-search($\boldsymbol{A}$, $\boldsymbol{P}$):
    $p$ ← 0        # number of $\boldsymbol{P}$
    $a_1$ ← 0       # number of sub-string of $\boldsymbol{A}$ at index 1

    for $i \leftarrow 1$ to $m$:
        $p \leftarrow \boldsymbol{P}[i] + d \times p$
        $a_1 \leftarrow \boldsymbol{A}[i] + d \times a_1$

    for $i \leftarrow 1$ to $n - m + 1$:
        if $i > 1$:
            $a_i \leftarrow d \times (a_{i-1} - \boxed{d^{m-1}} \times \boldsymbol{A}[i-1]) + \boldsymbol{A}[i+m-1]$
        if $p$ == $a_i$:
            output "there is a matching at $\boldsymbol{A}[i]$"
```

Precompute this before the for-loop in $O(\log m)$

- Time complexity is $O(m + n)$

String As Numbers (8)

❑ Limitation of SAN-search

- If $|\mathbf{\Sigma}|$ and m are large, then the converted numbers p and a_i are highly likely to overflow!
  - e.g., if $m = 40$, then we cannot represent 10^{m-1} as an integer variable

- What if we use a data structure called **big integer**?
  - Then, those numbers can be represented, but their arithmetic operations are not constant anymore ⇒ **not good** 😢

- How to resolve this issue?

Outline

❑ String matching

❑ Naïve algorithm

❑ SAN algorithm

❑ Rabin-Karp algorithm

Rabin-Karp Algorithm (1)

❑ Main idea to resolve big numbers

- Let's hash the numbers p and a_i into small numbers
  - Hashed numbers should be represented as a primitive type like `int`
  - A hashed number is called **fingerprint (FP)**
- If the FPs are the same, compare their original strings
  - Although the originals are different, their FPs can be the same (false match)
  - But, if the originals are the same, their FPs must be the same (true match)
- The FPs are computed with a large prime number $q \gg m$ as:
  - Pattern's fingerprint: $\tilde{p} = p \bmod q$
  - Document's fingerprint: $\tilde{a}_i = a_i \bmod q$

Rabin-Karp Algorithm (2)

❑ How to efficiently compute the fingerprints?

- There could be overflow during the computation of p
- How to obtain $p \bmod q$ with avoiding overflow?
  - $[(7 + 10(5 + 12345)] \bmod 9$
    - Note that $5 + 12345 = 1372 \times 9 + 2$
  - $\Rightarrow [7 + 10(1372 \times 9 + 2)] \bmod 9$
    - Note that 1372×9 does not affect the modulo operation
  - $\Rightarrow [7 + 10 \times 2] \bmod 9$
  - $\Rightarrow \left[7 + 10\left((5 + 12345) \bmod 9\right)\right] \bmod 9$
    - **Injecting "mod 9" into a large inner term does not affect the result!**
- Thus, we can avoid such overflow by injecting the modulo operation into an overflow-able term

Rabin-Karp Algorithm (3)

❑ How to efficiently compute the fingerprints?

- Now, let's consider the length of a pattern is 3

$$\tilde{p} = p \bmod q = [\boldsymbol{P}[3] + 10(\boldsymbol{P}[2] + 10\boldsymbol{P}[1])] \bmod q$$

- As described before, "`mod q`" is injected

$$\Rightarrow \tilde{p} = \big[\boldsymbol{P}[3] + 10\big((\boldsymbol{P}[2] + 10\boldsymbol{P}[1]) \bmod q\big)\big] \bmod q$$

- Initially, set $\tilde{p}$ to 0, and the final $\tilde{p}$ is computed as follows:

$$\tilde{p} \leftarrow (\boldsymbol{P}[1] + 10 \times \tilde{p}) \bmod q$$

$$\tilde{p} \leftarrow (\boldsymbol{P}[2] + 10 \times \tilde{p}) \bmod q$$

$$\tilde{p} \leftarrow (\boldsymbol{P}[3] + 10 \times \tilde{p}) \bmod q$$

```
$\tilde{p}$ ← 0
for $i \leftarrow 1$ to 3 ($\Rightarrow m$):
    $\tilde{p}$ ← $(\boldsymbol{P}[i] + 10 \times \tilde{p}) \bmod q$
```

Computing the fingerprint of a string of length m takes $O(m)$ time

Rabin-Karp Algorithm (4)

❑ Incremental update rule for a_i

- a_i is the number from a substring $\boldsymbol{A}[i \cdots i + m - 1]$

- Now, we need to obtain the fingerprint $\tilde{a}_i = a_i \bmod q$

$$a_i \bmod q = \left[10 \times (a_{i-1} - 10^{m-1} \times \boldsymbol{A}[i-1]) + \boldsymbol{A}[i + m - 1]\right] \bmod q$$

  - What is an overflow-able term? ⇒ 10^{m-1} (inject "mod" into this)
  - Is it Okay if we inject "mod" into a_{i-1}? ⇒ Yes, $a_{i-1} \bmod q = \tilde{a}_{i-1}$

$$a_i \bmod q = \left[10 \times (\tilde{a}_{i-1} - (10^{m-1} \bmod q) \times \boldsymbol{A}[i-1]) + \boldsymbol{A}[i + m - 1]\right] \bmod q$$

  - $\tilde{t} = 10^{m-1} \bmod q$ is obtained by $\tilde{t} \leftarrow (10 \times \tilde{t}) \bmod q$ for $i \leftarrow 1$ to $m - 1$ where $\tilde{t}$ is 1 initially
  - For a general string, consider base-$|\boldsymbol{\Sigma}|$ number system ($10 \Rightarrow |\boldsymbol{\Sigma}| = d$)

Rabin-Karp Algorithm (5)

❑ Pseudocode

n is the length of $\boldsymbol{A}$
m is the length of $\boldsymbol{P}$
d is $|\boldsymbol{\Sigma}|$

Precompute this before the for-loop (need to successively apply "mod" to avoid overflow)

```
def RK-search($\boldsymbol{A}$, $\boldsymbol{P}$, $q$):  # $q$ is a sufficiently large prime number
    $\tilde{p}$ ← 0        # fingerprint of the number of $\boldsymbol{P}$
    $\tilde{a}_1$ ← 0       # fingerprint of the number of substring of $\boldsymbol{A}$ at index 1

    for $i \leftarrow 1$ to $m$:
        $\tilde{p} \leftarrow (\boldsymbol{P}[i] + d \times p) \bmod q$
        $\tilde{a}_1 \leftarrow (\boldsymbol{A}[i] + d \times \tilde{a}_1) \bmod q$

    for $i \leftarrow 1$ to $n - m + 1$:
        if $i > 1$:
            $\tilde{a}_i \leftarrow \left[d \times (\tilde{a}_{i-1} - \boxed{(d^{m-1} \bmod q)} \times \boldsymbol{A}[i-1]) + \boldsymbol{A}[i+m-1]\right] \bmod q$

        if $\tilde{p}$ == $\tilde{a}_i$:
            if $\boldsymbol{P}[1 \cdots m]$ == $\boldsymbol{A}[i \cdots i+m-1]$:
# true match        output "there is a matching at $\boldsymbol{A}[i]$"
            else:
# false match       warn "just fingerprints are matched, not their originals"
```

Rabin-Karp Algorithm (6)

n is the length of $\boldsymbol{A}$
m is the length of $\boldsymbol{P}$
d is $|\boldsymbol{\Sigma}|$

❑ Time complexity analysis

```
def RK-search($\boldsymbol{A}$, $\boldsymbol{P}$, $q$):
    $\tilde{p} \leftarrow 0$
    $\tilde{a}_1 \leftarrow 0$

    for $i \leftarrow 1$ to $m$:                                   ⎫
        $\tilde{p} \leftarrow (\boldsymbol{P}[i] + d \times p) \bmod q$          ⎬ $O(m)$
        $\tilde{a}_1 \leftarrow (\boldsymbol{A}[i] + d \times \tilde{a}_1) \bmod q$       ⎭

    for $i \leftarrow 1$ to $n - m + 1$: ⇐ $O(n)$ repeats
        if $i > 1$:
            $\tilde{a}_i \leftarrow \left[d \times (\tilde{a}_{i-1} - (d^{m-1} \bmod q) \times \boldsymbol{A}[i-1]) + \boldsymbol{A}[i+m-1]\right] \bmod q$

        if $\tilde{p}$ == $\tilde{a}_i$: ⇐ Let $F$ be # of that FPs are hit
            if $\boldsymbol{P}[1 \cdots m]$ == $\boldsymbol{A}[i \cdots i+m-1]$: ⇐ $O(m)$
  # true match    output “there is a matching at $\boldsymbol{A}[i]$”
            else:
  # false match   warn “just fingerprints are matched, not their originals”
```

Total time complexity is
$O(n + Fm)$

Rabin-Karp Algorithm (7)

❑ Worst-case time complexity

- The worst-case of RK algorithm is when $F = n$
  - e.g., if $\boldsymbol{A}$ = “$aaaaaaaa$” and $\boldsymbol{P}$ = “aaa”, their FPs are hit for each iteration
- In this case, the time complexity is $\mathrm{O}(n + Fm) = \mathrm{O}(nm)$
  - Not improved compared to Naïve algorithm

❑ Average-case time complexity

- If characters are uniformly distributed, $\mathrm{P}(\tilde{p} = \tilde{a}_i) = 1/q$
  - Because the range of a FP is between 0 and $q - 1$.
- On average, $F = n/q$ for about n tries of FP comparisons
- If we pick a large $q \gg m$, then $Fm = \frac{m}{q}n = cn$ where $c \leq 1$.
- In this case, the time complexity is $\mathrm{O}(n + Fm) = \mathrm{O}(n)$

What You Need To Know

❑ String matching

- Search for a pattern string in a document string

❑ String as numbers

- Rephrase string matching to number matching
- Convert a string to a number in base-$|\boldsymbol{\Sigma}|$ system
- Basic number matching approach has an overflow issue

❑ Rabin-Karp algorithm

- Let's hash the numbers p and a_i into small fingerprints
  - If their FPs are the same, compare their originals to avoid false match
- Time complexity is $O(n + Fm)$
  - Worst: $O(nm)$, Average: $O(n)$

In Next Lecture

- ❑ More efficient string search algorithms
  - ▪ Automata algorithm

Thank You