Analysis of Algorithms

String matching

Andres Mendez-Vazquez

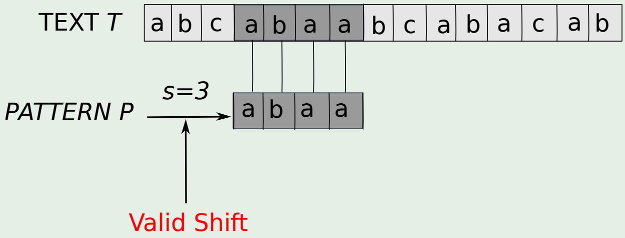
November 24, 2015

Outline

1. String Matching
   - Introduction
   - Some Notation
2. Naive Algorithm
   - Using Brute Force
3. The Rabin-Karp Algorithm
   - Efficiency After All
   - Horner's Rule
   - Generaiting Possible Matches
   - The Final Algorithm
   - Other Methods
4. Exercises

What is string matching?

Given two sequences of characters drawn from a finite alphabet Σ, $T[1..n]$ and $P[1..m]$



Where...

A Valid Shift

P occurs with a ***valid shift*** s if for some $0 \leq s \leq n - m \Longrightarrow T[s+1..s+m] == P[1..m]$.

Otherwise

it is an invalid shift.

Thus

The string-matching problem is the problem of of finding all valid shifts given a patten P on a text T.

Where...

A Valid Shift

P occurs with a ***valid shift*** s if for some $0 \leq s \leq n - m \Longrightarrow T[s+1..s+m] == P[1..m]$.

Otherwise

it is an invalid shift.

Thus

The string-matching problem is the problem of of finding all valid shifts given a patten P on a text T.

Where...

A Valid Shift

P occurs with a ***valid shift*** s if for some $0 \leq s \leq n - m \Longrightarrow T[s+1..s+m] == P[1..m]$.

Otherwise

it is an invalid shift.

Thus

The string-matching problem is the problem of of finding all valid shifts given a patten P on a text T.

Possible Algorithms

We have the following ones

Algorithm	Preprocessing Time	Worst Case Matching Time
Naive	0	$O\left(\left(n-m+1\right)m\right)$
Rabin-Karp	$\Theta\left(m\right)$	$O\left(\left(n-m+1\right)m\right)$
Finite Automaton	$O\left(m\left\|\Sigma\right\|\right)$	$\Theta\left(n\right)$
Knuth-Morris-Pratt	$\Theta\left(m\right)$	$\Theta\left(n\right)$

Outline

1. String Matching
   - Introduction
   - Some Notation
2. Naive Algorithm
   - Using Brute Force
3. The Rabin-Karp Algorithm
   - Efficiency After All
   - Horner's Rule
   - Generaiting Possible Matches
   - The Final Algorithm
   - Other Methods
4. Exercises

Notation and Terminology

Definition

We denote by Σ^* (read "sigma-star") the set of all finite length strings formed using characters from the alphabet Σ.

Notation and Terminology

Definition

We denote by Σ^* (read "sigma-star") the set of all finite length strings formed using characters from the alphabet Σ.

Constraint

We assume a finite length strings.

- The length of a string x is denoted $|x|$.
- The concatenation of two strings x and y, denoted xy, has length $|x| + |y|$.

Notation and Terminology

Definition

We denote by Σ^* (read "sigma-star") the set of all finite length strings formed using characters from the alphabet Σ.

Constraint

We assume a finite length strings.

Some basic concepts

- The zero **empty string**, ϵ, also belong to Σ^*.
- The length of a string x is denoted $|x|$.
- The concatenation of two strings x and y, denoted xy, has length $|x| + |y|$

Notation and Terminology

Definition

We denote by Σ^* (read "sigma-star") the set of all finite length strings formed using characters from the alphabet Σ.

Constraint

We assume a finite length strings.

Some basic concepts

- The zero **empty string**, ϵ, also belong to Σ^*.
- The length of a string x is denoted $|x|$.
- The concatenation of two strings x and y, denoted xy, has length $|x| + |y|$

Notation and Terminology

Definition

We denote by Σ^* (read "sigma-star") the set of all finite length strings formed using characters from the alphabet Σ.

Constraint

We assume a finite length strings.

Some basic concepts

- The zero **empty string**, ϵ, also belong to Σ^*.
- The length of a string x is denoted $|x|$.
- The concatenation of two strings x and y, denoted xy, has length $|x| + |y|$

Notation and Terminology

Prefix

A string w is a prefix x, $w \sqsubset x$ if $x = wy$ for some string $y \in \Sigma^*$.

Notation and Terminology

Prefix

A string w is a prefix x, $w \sqsubset x$ if $x = wy$ for some string $y \in \Sigma^*$.

Suffix

A string w is a suffix x, $w \sqsupset x$ if $x = yw$ for some string $y \in \Sigma^*$.

- Suffix: If $w \sqsupset x \Rightarrow |w| \leq |x|$
- The ε is both suffix and prefix of every string.

Notation and Terminology

Prefix

A string w is a prefix x, $w \sqsubset x$ if $x = wy$ for some string $y \in \Sigma^*$.

Suffix

A string w is a suffix x, $w \sqsupset x$ if $x = yw$ for some string $y \in \Sigma^*$.

Properties

- Prefix: If $w \sqsubset x \Rightarrow |w| \leq |x|$
- Suffix: If $w \sqsupset x \Rightarrow |w| \leq |x|$
- The ϵ is both suffix and prefix of every string.

Notation and Terminology

Prefix

A string w is a prefix x, $w \sqsubset x$ if $x = wy$ for some string $y \in \Sigma^*$.

Suffix

A string w is a suffix x, $w \sqsupset x$ if $x = yw$ for some string $y \in \Sigma^*$.

Properties

- Prefix: If $w \sqsubset x \Rightarrow |w| \leq |x|$
- Suffix: If $w \sqsupset x \Rightarrow |w| \leq |x|$
- The ϵ is both suffix and prefix of every string.

Notation and Terminology

Prefix

A string w is a prefix x, $w \sqsubset x$ if $x = wy$ for some string $y \in \Sigma^*$.

Suffix

A string w is a suffix x, $w \sqsupset x$ if $x = yw$ for some string $y \in \Sigma^*$.

Properties

- Prefix: If $w \sqsubset x \Rightarrow |w| \leq |x|$
- Suffix: If $w \sqsupset x \Rightarrow |w| \leq |x|$
- The ϵ is both suffix and prefix of every string.

Notation and Terminology

In addition

- For any string x and y and any character a, we have $w \sqsupset x$ if and only if $aw \sqsupset ax$
- In addition, $\sqsubset$ and $\sqsupset$ are transitive relations.

Notation and Terminology

In addition

- For any string x and y and any character a, we have $w \sqsupset x$ if and only if $aw \sqsupset ax$
- In addition, $\sqsubset$ and $\sqsupset$ are transitive relations.

Outline

1. String Matching
   - Introduction
   - Some Notation
2. Naive Algorithm
   - Using Brute Force
3. The Rabin-Karp Algorithm
   - Efficiency After All
   - Horner's Rule
   - Generaiting Possible Matches
   - The Final Algorithm
   - Other Methods
4. Exercises

The naive string algorithm

Algorithm

NAIVE-STRING-MATCHER(T, P)

1. $n = T.length$
2. $m = P.length$
3. for $s = 0$ to $n - m$
4.   if $P[1..m] == T[s+1..s+m]$
5.   print "Pattern occurs with shift" s

Complexity

$O((n-m+1)m)$ or $\Theta(n^2)$ if $m = \lfloor \frac{n}{2} \rfloor$

The naive string algorithm

Algorithm

NAIVE-STRING-MATCHER(T, P)

1. $n = T.length$
2. $m = P.length$
3. for $s = 0$ to $n - m$
4.   if $P[1..m] == T[s+1..s+m]$
5.   print "Pattern occurs with shift" s

Complexity

$O((n-m+1)m)$ or $\Theta(n^2)$ if $m = \lfloor \frac{n}{2} \rfloor$

Outline

1 String Matching
- Introduction
- Some Notation

2 Naive Algorithm
- Using Brute Force

3 The Rabin-Karp Algorithm
- Efficiency After All
- Horner's Rule
- Generaiting Possible Matches
- The Final Algorithm
- Other Methods

4 Exercises

A more elaborated algorithm

Rabin-Karp algorithm

Lets assume that $\Sigma = \{0, 1, ..., 9\}$ then we have the following:

- Thus, each string of k consecutive characters is a k-length decimal number:

$$c_1 c_2 \cdots c_{k-1} c_k = 10^{k-1} c_1 + 10^{k-2} c_2 + \ldots + 10 c_{k-1} + c_k$$

A more elaborated algorithm

Rabin-Karp algorithm

Lets assume that $\Sigma = \{0, 1, ..., 9\}$ then we have the following:

- Thus, each string of k consecutive characters is a k-length decimal number:
  $c_1 c_2 \cdots c_{k-1} c_k = 10^{k-1} c_1 + 10^{k-2} c_2 + \ldots + 10 c_{k-1} + c_k$

Thus

- p correspond the decimal number for pattern $P[1..m]$.
- t_s denote decimal value of m-length substring $T[s+1..s+m]$ for $s = 0, 1, ..., n-m$.

A more elaborated algorithm

Rabin-Karp algorithm

Lets assume that $\Sigma = \{0, 1, ..., 9\}$ then we have the following:

- Thus, each string of k consecutive characters is a k-length decimal number:
  $c_1 c_2 \cdots c_{k-1} c_k = 10^{k-1} c_1 + 10^{k-2} c_2 + \ldots + 10 c_{k-1} + c_k$

Thus

- p correspond the decimal number for pattern $P[1..m]$.
- t_s denote decimal value of m-length substring $T[s+1..s+m]$ for $s = 0, 1, ..., n-m$.

A more elaborated algorithm

Rabin-Karp algorithm

Lets assume that $\Sigma = \{0, 1, ..., 9\}$ then we have the following:

- Thus, each string of k consecutive characters is a k-length decimal number:
  $c_1 c_2 \cdots c_{k-1} c_k = 10^{k-1} c_1 + 10^{k-2} c_2 + \ldots + 10 c_{k-1} + c_k$

Thus

- p correspond the decimal number for pattern $P[1..m]$.
- t_s denote decimal value of m-length substring $T[s+1..s+m]$ for $s = 0, 1, ..., n-m$.

Then

Properties

Clearly $t_s == p$ if and only if $T[s+1..s+m] == P[1..m]$, thus s is a valid shift.

Now, think about this

What if we put everything in a single word of the machine

- If we can compute p in $\Theta(m)$.
- If we can compute all the t_s in $\Theta(n-m+1)$.

Now, think about this

What if we put everything in a single word of the machine

- If we can compute p in $\Theta\left(m\right)$.
- If we can compute all the t_s in $\Theta\left(n-m+1\right)$.

We will get

$\Theta\left(m\right)+\Theta\left(n-m+1\right)=\Theta\left(n\right)$

Now, think about this

What if we put everything in a single word of the machine

- If we can compute p in $\Theta(m)$.
- If we can compute all the t_s in $\Theta(n-m+1)$.

We will get

$\Theta(m) + \Theta(n-m+1) = \Theta(n)$

Outline

1. String Matching
   - Introduction
   - Some Notation
2. Naive Algorithm
   - Using Brute Force
3. The Rabin-Karp Algorithm
   - Efficiency After All
   - Horner's Rule
   - Generaiting Possible Matches
   - The Final Algorithm
   - Other Methods
4. Exercises

Remember Horner's Rule

Consider

- We can compute, the decimal representation by the Horner's rule:

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + ... + 10(P[2] + 10P[1])...)) \tag{1}$$

Thus, we can compute t_0 using this rule in

$$\Theta(m) \tag{2}$$

Remember Horner's Rule

Consider

- We can compute, the decimal representation by the Horner's rule:

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + ... + 10(P[2] + 10P[1])...)) \quad (1)$$

Thus, we can compute t_0 using this rule in

$$\Theta(m) \quad (2)$$

Example

If you have the following set of digits

2	4	0	1

Using the Horner's Rule for $m = 4$

$$
\begin{aligned}
2401 &= 1 + 10 \times (0 + 10 \times (4 + 10 \times 2)) \\
&= 2 \times 10^3 + 4 \times 10^2 + 0 \times 10 + 1
\end{aligned}
$$

Example

If you have the following set of digits

2	4	0	1

Using the Horner's Rule for $m = 4$

$$\begin{aligned} 2401 &= 1 + 10 \times (0 + 10 \times (4 + 10 \times 2)) \\ &= 2 \times 10^3 + 4 \times 10^2 + 0 \times 10 + 1 \end{aligned}$$

Then

To compute the remaining values, we can use the previous value

$$t_{s+1} = 10\left(t_s - 10^{m-1}T\left[s+1\right]\right) + T\left[s+m+1\right]. \quad (3)$$

Notice the following

- Subtracting from it $10^{m-1}T\left[s+1\right]$ removes the high-order digit from t_s.

Then

To compute the remaining values, we can use the previous value

$$t_{s+1} = 10\left(t_s - 10^{m-1}T[s+1]\right) + T[s+m+1]. \quad (3)$$

Notice the following

- Subtracting from it $10^{m-1}T[s+1]$ removes the high-order digit from t_s.
- Multiplying the result by 10 shifts the number left by one digit position.
- Adding $T[s+m+1]$ brings in the appropriate low-order digit.

Then

To compute the remaining values, we can use the previous value

$$t_{s+1} = 10\left(t_s - 10^{m-1}T[s+1]\right) + T[s+m+1]. \quad (3)$$

Notice the following

- Subtracting from it $10^{m-1}T[s+1]$ removes the high-order digit from t_s.
- Multiplying the result by 10 shifts the number left by one digit position.
- Adding $T[s+m+1]$ brings in the appropriate low-order digit.

Example

Imagine that you have...

index	1	2	3	4	5	6
digit	1	0	2	4	1	0

1. We have $t_0 = 1024$ then we want to calculate 0241
2. Then we subtract $\left(10^3 \times T[1]\right) == 1000$ of 1024
3. We get 0024
4. Multiply by 10, and we get 0240
5. We add $T[5] == 1$
6. Finally, we get 0241

Example

Imagine that you have...

index	1	2	3	4	5	6
digit	1	0	2	4	1	0

1. We have $t_0 = 1024$ then we want to calculate 0241
2. Then we subtract $(10^3 \times T[1]) == 1000$ of 1024
3. We get 0024
4. Multiply by 10, and we get 0240
5. We add $T[5] == 1$
6. Finally, we get 0241

Example

Imagine that you have...

index	1	2	3	4	5	6
digit	1	0	2	4	1	0

1. We have $t_0 = 1024$ then we want to calculate 0241
2. Then we subtract $\left(10^3 \times T\left[1\right]\right) == 1000$ of 1024
3. We get 0024
4. Multiply by 10, and we get 0240
5. We add $T\left[5\right] == 1$
6. Finally, we get 0241

Example

Imagine that you have...

index	1	2	3	4	5	6
digit	1	0	2	4	1	0

1. We have $t_0 = 1024$ then we want to calculate 0241
2. Then we subtract $(10^3 \times T[1]) == 1000$ of 1024
3. We get 0024
4. Multiply by 10, and we get 0240
5. We add $T[5] == 1$
6. Finally, we get 0241

Example

Imagine that you have...

index	1	2	3	4	5	6
digit	1	0	2	4	1	0

1. We have $t_0 = 1024$ then we want to calculate 0241
2. Then we subtract $\left(10^3 \times T[1]\right) == 1000$ of 1024
3. We get 0024
4. Multiply by 10, and we get 0240
5. We add $T[5] == 1$
6. Finally, we get 0241

Example

Imagine that you have...

index	1	2	3	4	5	6
digit	1	0	2	4	1	0

1. We have $t_0 = 1024$ then we want to calculate 0241
2. Then we subtract $(10^3 \times T[1]) == 1000$ of 1024
3. We get 0024
4. Multiply by 10, and we get 0240
5. We add $T[5] == 1$
6. Finally, we get 0241

Example

Imagine that you have...

index	1	2	3	4	5	6
digit	1	0	2	4	1	0

1. We have $t_0 = 1024$ then we want to calculate 0241
2. Then we subtract $(10^3 \times T[1]) == 1000$ of 1024
3. We get 0024
4. Multiply by 10, and we get 0240
5. We add $T[5] == 1$
6. Finally, we get 0241

Remarks

First

We can extend this beyond the decimals to any d digit system!!!

What happens when the numbers are quite large?

We can use the module

Meaning

- Compute p and t_s values modulo a suitable modulus q.

Remarks

First

We can extend this beyond the decimals to any d digit system!!!

What happens when the numbers are quite large?

We can use the module

Meaning

- Compute p and t_s values modulo a suitable modulus q.

Remarks

First

We can extend this beyond the decimals to any d digit system!!!

What happens when the numbers are quite large?

We can use the module

Meaning

- Compute p and t_s values modulo a suitable modulus q.

Outline

1. String Matching
   - Introduction
   - Some Notation

2. Naive Algorithm
   - Using Brute Force

3. The Rabin-Karp Algorithm
   - Efficiency After All
   - Horner's Rule
   - Generaiting Possible Matches
   - The Final Algorithm
   - Other Methods

4. Exercises

Remember Hash Functions

Yes, we are mapping the large numbers into the set

$$\{0, 1, 2, ..., q - 1\}$$

Then, to reduce the possible representation

Use the module of q

It is possible to compute $p \mod q$ in $\Theta(m)$ time and all the $t_s \mod q$ in $\Theta(n-m+1)$ time.

Something Notable

If we choose the modulus q as a prime such that $10q$ just fits within one computer word, then we can perform all the necessary computations with single-precision arithmetic.

Then, to reduce the possible representation

Use the module of q

It is possible to compute $p \mod q$ in $\Theta(m)$ time and all the $t_s \mod q$ in $\Theta(n-m+1)$ time.

Something Notable

If we choose the modulus q as a prime such that $10q$ just fits within one computer word, then we can perform all the necessary computations with single-precision arithmetic.

Why $10q$?

After all

$10q$ is the number that will subtracting for or multiplying by!!!

We use "truncated division" to implement the modulo operation

For example given two numbers a and n, we can do the following

$$q = trunc\left(\frac{a}{n}\right)$$

Then

$$r = a - nq$$

Why $10q$?

After all

$10q$ is the number that will subtracting for or multiplying by!!!

We use "truncated division" to implement the modulo operation

For example given two numbers a and n, we can do the following

$$q = trunc\left(\frac{a}{n}\right)$$

Then

$$r = a - nq$$

Why $10q$?

After all

$10q$ is the number that will subtracting for or multiplying by!!!

We use "truncated division" to implement the modulo operation

For example given two numbers a and n, we can do the following

$$q = trunc\left(\frac{a}{n}\right)$$

Then

$$r = a - nq$$

Why $10q$?

Thus

If q is a prime we can use this as the element of truncated division then $r = a - 10q$.

Truncated Algorithm

Truncated-Module(a, q)

$r = a - 10q$

while $r > 10q$

$\qquad r = r - 10q$

return r

Why $10q$?

Thus

If q is a prime we can use this as the element of truncated division then $r = a - 10q$.

Truncated Algorithm

Truncated-Module(a, q)

1. $r = a - 10q$
2. while $r > 10q$
3. $\quad r = r - 10q$
4. return r

Then

Thus

Then, we can implement this in basic arithmetic CPU operations.

Not only that

In general, for a d-ary alphabet, we choose q such that dq fits within a computer word.

Then

Thus

Then, we can implement this in basic arithmetic CPU operations.

Not only that

In general, for a d-ary alphabet, we choose q such that dq fits within a computer word.

In general, we have

The following

$$t_{s+1} = \left(d\left(t_s - T[s+1]h\right) + T[s+m+1]\right) \mod q \quad (4)$$

where $h \equiv d^{m-1} (\text{mod } q)$ is the value of the digit "1" in the high-order position of an m-digit text window.

Here

We have a small problem!!!

Question?

Can we differentiate between $p \mod q$ and $t_s \mod q$?

In general, we have

The following

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \mod q \quad (4)$$

where $h \equiv d^{m-1} (\text{mod } q)$ is the value of the digit "1" in the high-order position of an m-digit text window.

Here

We have a small problem!!!

Question?

Can we differentiate between $p \mod q$ and $t_s \mod q$?

In general, we have

The following

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \mod q \quad (4)$$

where $h \equiv d^{m-1} (\text{mod } q)$ is the value of the digit "1" in the high-order position of an m-digit text window.

Here

We have a small problem!!!

Question?

Can we differentiate between $p \mod q$ and $t_s \mod q$?

What?

Look at this with $q = 11$

- $14 \mod 11 == 3$
- $25 \mod 11 == 3$

But $14 \neq 25$

What?

Look at this with $q = 11$

- $14 \mod 11 == 3$
- $25 \mod 11 == 3$

But $14 \neq 25$

However

- $11 \mod 11 == 0$
- $25 \mod 11 == 3$

We can say that $11 \neq 25$!!!

What?

Look at this with $q = 11$

- $14 \mod 11 == 3$
- $25 \mod 11 == 3$

But $14 \neq 25$

However

- $11 \mod 11 == 0$
- $25 \mod 11 == 3$

We can say that $11 \neq 25$!!!

This means

We can use the modulo to differentiate numbers, but not to exactly to say if they are equal!!!

What?

Look at this with $q = 11$

- $14 \mod 11 == 3$
- $25 \mod 11 == 3$

But $14 \neq 25$

However

- $11 \mod 11 == 0$
- $25 \mod 11 == 3$

We can say that $11 \neq 25$!!!

This means

We can use the modulo to differentiate numbers, but not to exactly to say if they are equal!!!

What?

Look at this with $q = 11$

- $14 \mod 11 == 3$
- $25 \mod 11 == 3$

But $14 \neq 25$

However

- $11 \mod 11 == 0$
- $25 \mod 11 == 3$

We can say that $11 \neq 25$!!!

This means

We can use the modulo to differentiate numbers, but not to exactly to say if they are equal!!!

What?

Look at this with $q = 11$

- $14 \mod 11 == 3$
- $25 \mod 11 == 3$

But $14 \neq 25$

However

- $11 \mod 11 == 0$
- $25 \mod 11 == 3$

We can say that $11 \neq 25$!!!

This means

We can use the modulo to differentiate numbers, but not to exactly to say if they are equal!!!

What?

Look at this with $q = 11$

- $14 \mod 11 == 3$
- $25 \mod 11 == 3$

But $14 \neq 25$

However

- $11 \mod 11 == 0$
- $25 \mod 11 == 3$

We can say that $11 \neq 25$!!!

This means

We can use the modulo to differentiate numbers, but not to exactly to say if they are equal!!!

Thus

We have the following logic

- If $t_s \equiv p \pmod{q}$ does not mean that $t_s == p$.
- If $t_s \not\equiv p \pmod{q}$, we have that $t_s \neq p$.

Thus

We have the following logic

- If $t_s \equiv p \pmod{q}$ does not mean that $t_s == p$.
- If $t_s \not\equiv p \pmod{q}$, we have that $t_s \neq p$.

Fixing the problem

To fix this problem we simply test to see if the hit is not spurious.

Thus

We have the following logic

- If $t_s \equiv p \pmod{q}$ does not mean that $t_s == p$.
- If $t_s \not\equiv p \pmod{q}$, we have that $t_s \neq p$.

Fixing the problem

To fix this problem we simply test to see if the hit is not spurious.

Note

If q is large enough, then we can hope that spurious hits occur infrequently enough that the cost of the extra checking is low.

cinvestav
Unidad Guadalajara

Thus

We have the following logic

- If $t_s \equiv p \pmod{q}$ does not mean that $t_s == p$.
- If $t_s \not\equiv p \pmod{q}$, we have that $t_s \neq p$.

Fixing the problem

To fix this problem we simply test to see if the hit is not spurious.

Note

If q is large enough, then we can hope that spurious hits occur infrequently enough that the cost of the extra checking is low.

Outline

1. String Matching
   - Introduction
   - Some Notation
2. Naive Algorithm
   - Using Brute Force
3. The Rabin-Karp Algorithm
   - Efficiency After All
   - Horner's Rule
   - Generaiting Possible Matches
   - The Final Algorithm
   - Other Methods
4. Exercises

The Final Algorithm

Rabin-Karp-Matcher(T, P, d, q)

$n = T.length$

$m = P.length$

$h = d^{m-1} \mod q$ // Storing the reminder of the highest power

$p = 0$

$t_0 = 0$

for $i = 1$ **to** m // Preprocessing

    $p = (dp + P[i]) \mod q$

    $t_0 = (dp + T[i]) \mod q$

for $s = 0$ **to** $n - m$

    if $p == t_s$

        if $P[1..m] == T[s+1..s+m]$ // Actually a Loop

            print "Pattern occurs with shift" s

    if $s < n - m$

        $t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \mod q$

The Final Algorithm

Rabin-Karp-Matcher(T, P, d, q)

```
n = T.length
m = P.length
h = d^(m-1) mod q // Storing the reminder of the highest power
p = 0
t_0 = 0
for i = 1 to m // Preprocessing
    p = (dp + P[i]) mod q
    t_0 = (dp + T[i]) mod q
for s = 0 to n - m
    if p == t_s
        if P[1..m] == T[s+1..s+m] // Actually a Loop
            print "Pattern occurs with shift" s
    if s < n - m
        t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) mod q
```

The Final Algorithm

Rabin-Karp-Matcher(T, P, d, q)

$n = T.length$

$m = P.length$

$h = d^{m-1} \mod q$ // Storing the reminder of the highest power

$p = 0$

$t_0 = 0$

for $i = 1$ **to** m // Preprocessing

    $p = (dp + P[i]) \mod q$

    $t_0 = (dp + T[i]) \mod q$

for $s = 0$ **to** $n - m$

    if $p == t_s$

        if $P[1..m] == T[s+1..s+m]$ // Actually a Loop

            print "Pattern occurs with shift" s

    if $s < n - m$

        $t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \mod q$

The Final Algorithm

Rabin-Karp-Matcher(T, P, d, q)

$n = T.length$

$m = P.length$

$h = d^{m-1} \mod q$ // Storing the reminder of the highest power

$p = 0$

$t_0 = 0$

for $i = 1$ **to** m // Preprocessing

  $p = (dp + P[i]) \mod q$

  $t_0 = (dp + T[i]) \mod q$

for $s = 0$ **to** $n - m$

  if $p == t_s$

    if $P[1..m] == T[s+1..s+m]$ // Actually a Loop

      print "Pattern occurs with shift" s

  if $s < n - m$

    $t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \mod q$

Complexity

Preprocessing

1. $p \mod q$ is done in $\Theta(m)$.
2. $t'_s s \mod q$ is done in $\Theta(n-m+1)$.

Then checking $P[1..m] == T[s+1..s+m]$

In the worst case, $\Theta((n-m+1)m)$

Complexity

Preprocessing

1. $p \mod q$ is done in $\Theta(m)$.
2. $t_s' s \mod q$ is done in $\Theta(n-m+1)$.

Then checking $P[1..m] == T[s+1..s+m]$

In the worst case, $\Theta((n-m+1)m)$

Still we can do better!

First

The number of spurious hits is $O\left(n/q\right)$.

Because

We can estimate the chance that an arbitrary t_s will be equivalent to $p \mod q$ as $1/q$.

Properties

Since there are $O\left(n\right)$ positions at which the test of line 10 fails (Thus, you have $O\left(n/q\right)$ non valid hits) and we spend $O\left(m\right)$ time per hit

Still we can do better!

First

The number of spurious hits is $O\left(n/q\right)$.

Because

We can estimate the chance that an arbitrary t_s will be equivalent to $p \mod q$ as $1/q$.

Properties

Since there are $O\left(n\right)$ positions at which the test of line 10 fails (Thus, you have $O\left(n/q\right)$ non valid hits) and we spend $O\left(m\right)$ time per hit

Still we can do better!

First

The number of spurious hits is $O\left({}^{n}/_{q}\right)$.

Because

We can estimate the chance that an arbitrary t_s will be equivalent to $p \mod q$ as ${}^{1}/_{q}$.

Properties

Since there are $O(n)$ positions at which the test of line 10 fails (Thus, you have $O\left({}^{n}/_{q}\right)$ non valid hits) and we spend $O(m)$ time per hit

Finally, we have that

The expected matching time

The expected matching time by Rabin-Karp algorithm is

$$O(n) + O\left(m\left(v + \frac{n}{q}\right)\right)$$

where v is the number of valid shifts.

Finally, we have that

The expected matching time

The expected matching time by Rabin-Karp algorithm is

$$O(n) + O\left(m\left(v + \frac{n}{q}\right)\right)$$

where v is the number of valid shifts.

In addition

If $v = O(1)$ (Number of valid shifts small) and choose $q \geq m$ such that $\frac{n}{q} = O(1)$ (q to be larger enough than the pattern's length).

- Finally, because $m \leq n$, thus the expected time is $O(n)$.

Finally, we have that

The expected matching time

The expected matching time by Rabin-Karp algorithm is

$$O(n) + O\left(m\left(v + \frac{n}{q}\right)\right)$$

where v is the number of valid shifts.

In addition

If $v = O(1)$ (Number of valid shifts small) and choose $q \geq m$ such that $\frac{n}{q} = O(1)$ (q to be larger enough than the pattern's length).

Then

- The algorithm takes $O(m + n)$ for finding the matches.
- Finally, because $m \leq n$, thus the expected time is $O(n)$.

Finally, we have that

The expected matching time

The expected matching time by Rabin-Karp algorithm is

$$O(n) + O\left(m\left(v + \frac{n}{q}\right)\right)$$

where v is the number of valid shifts.

In addition

If $v = O(1)$ (Number of valid shifts small) and choose $q \geq m$ such that $\frac{n}{q} = O(1)$ (q to be larger enough than the pattern's length).

Then

- The algorithm takes $O(m+n)$ for finding the matches.
- Finally, because $m \leq n$, thus the expected time is $O(n)$.

Finally, we have that

The expected matching time

The expected matching time by Rabin-Karp algorithm is

$$O(n) + O\left(m\left(v + \frac{n}{q}\right)\right)$$

where v is the number of valid shifts.

In addition

If $v = O(1)$ (Number of valid shifts small) and choose $q \geq m$ such that $\frac{n}{q} = O(1)$ (q to be larger enough than the pattern's length).

Then

- The algorithm takes $O(m+n)$ for finding the matches.
- Finally, because $m \leq n$, thus the expected time is $O(n)$.

Outline

1. String Matching
   - Introduction
   - Some Notation
2. Naive Algorithm
   - Using Brute Force
3. The Rabin-Karp Algorithm
   - Efficiency After All
   - Horner's Rule
   - Generaiting Possible Matches
   - The Final Algorithm
   - Other Methods
4. Exercises

We have other methods

We have the following ones

Algorithm	Preprocessing Time	Worst Case Matching Time
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite Automaton	$O(m\|\Sigma\|)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

Remarks about Knuth-Morris-Pratt

The Algorithm

It is quite an elegant algorithm that improves over the state machine.

How?

It avoid to compute the transition function in the state machine by using the prefix function π

Remarks about Knuth-Morris-Pratt

The Algorithm

It is quite an elegant algorithm that improves over the state machine.

How?

It avoid to compute the transition function in the state machine by using the prefix function π

- It encapsulate information how the pattern matches against shifts of itself

However, At the same time (Circa 1977)

Boyer–Moore string search algorithm

It was presented at the same time

It is used in the GREP function for pattern matching in UNIX

Actually is the basic algorithm to beat when doing research in this area!!!

Richard Cole (Circa 1991)

He gave a a proof of the algorithm with an upper bound of $3m$ comparisons in the worst case!!!

However, At the same time (Circa 1977)

Boyer–Moore string search algorithm

It was presented at the same time

It is used in the GREP function for pattern matching in UNIX

Actually is the basic algorithm to beat when doing research in this area!!!

Richard Cole (Circa 1991)

He gave a a proof of the algorithm with an upper bound of $3m$ comparisons in the worst case!!!

However, At the same time (Circa 1977)

Boyer–Moore string search algorithm

It was presented at the same time

It is used in the GREP function for pattern matching in UNIX

Actually is the basic algorithm to beat when doing research in this area!!!

Richard Cole (Circa 1991)

He gave a a proof of the algorithm with an upper bound of $3m$ comparisons in the worst case!!!

Exercises

- 32.1-1
- 32.1-2
- 32.1-4
- 32.2-1
- 32.2-2
- 32.2-3
- 32.2-4