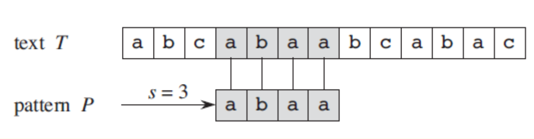
Introduction to Algorithms
Lecture 14 String Matching

Xue Chen
xuechen1989@ustc.edu.cn
2025 spring in

Outline

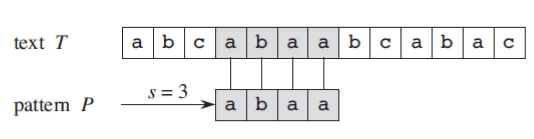
Introduction

Given Text $T[1\ldots n]$ and Pattern $P[1\ldots m]$, find all shifts s such that $T[s+1\ldots s+m] = P[1\ldots m]$



Introduction

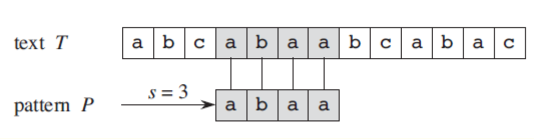
Given Text $T[1 \ldots n]$ and Pattern $P[1 \ldots m]$, find all shifts s such that $T[s+1 \ldots s+m] = P[1 \ldots m]$



1. Applications in bio-computation (DNA matching), search engine, document processing, ...

Introduction

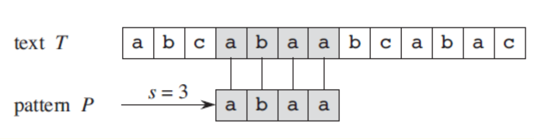
Given Text $T[1 \dots n]$ and Pattern $P[1 \dots m]$, find all shifts s such that $T[s+1 \dots s+m] = P[1 \dots m]$



1. Applications in bio-computation (DNA matching), search engine, document processing, ...
2. n and m are large, say $\geqslant 10^6$
3. Any idea of designing a string-matching algorithm?

Introduction

Given Text $T[1 \ldots n]$ and Pattern $P[1 \ldots m]$, find all shifts s such that $T[s+1 \ldots s+m] = P[1 \ldots m]$



1. Applications in bio-computation (DNA matching), search engine, document processing, ...
2. n and m are large, say $\geqslant 10^6$
3. Any idea of designing a string-matching algorithm?
4. Our goal is to have time $O(n+m)$

Preliminaries

1. Σ: Alphabet with $|\Sigma| = O(1)$ like $\{a, b, \ldots, z\}$ or $\{0, 1, \ldots, 9\}$
2. Σ^n: Strings of length n with alphabet Σ
3. $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \ldots$ denotes all strings of alphabet Σ

Preliminaries

1. Σ: Alphabet with $|\Sigma| = O(1)$ like $\{a, b, \ldots, z\}$ or $\{0, 1, \ldots, 9\}$
2. Σ^n: Strings of length n with alphabet Σ
3. $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \ldots$ denotes all strings of alphabet Σ
4. Prefix $w \sqsubset x$: String w is a prefix of string x, i.e., $x = wy$ for some $y \in \Sigma^*$
5. Suffix $y \sqsupset x$: String y is a suffix of string x, i.e., $x = wy$ for some $w \in \Sigma^*$

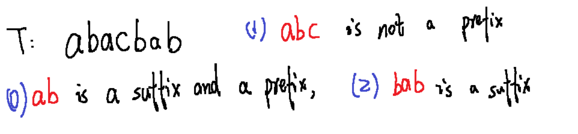
T: abacbab (1) abc is not a prefix

(0) ab is a suffix and a prefix, (2) bab is a suffix

Example: Examples of prefixes and suffixes

Preliminaries

1. Σ: Alphabet with $|\Sigma| = O(1)$ like $\{a, b, \ldots, z\}$ or $\{0, 1, \ldots, 9\}$
2. Σ^n: Strings of length n with alphabet Σ
3. $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \ldots$ denotes all strings of alphabet Σ
4. Prefix $w \sqsubset x$: String w is a prefix of string x, i.e., $x = wy$ for some $y \in \Sigma^*$
5. Suffix $y \sqsupset x$: String y is a suffix of string x, i.e., $x = wy$ for some $w \in \Sigma^*$



Example: Examples of prefixes and suffixes

6. Lemma 32.1 in CLRS: If $x \sqsupset z$ and $y \sqsupset z$, either $x \sqsupset y$ or $y \sqsupset$ depends on their length $|x|$ and $|y|$.

Outline

Introduction

Recall our problem: Given Text $T[1 \ldots n]$ and Pattern $P[1 \ldots m]$, find all shifts s such that $T[s+1 \ldots s+m] = P[1 \ldots m]$.

Basic Idea

When checking $T[s+1 \ldots s+m] = P[1 \ldots m]$, instead of comparing each element, compare their hash values

Introduction

Recall our problem: Given Text $T[1\dots n]$ and Pattern $P[1\dots m]$, find all shifts s such that $T[s+1\dots s+m] = P[1\dots m]$.

Basic Idea

When checking $T[s+1\dots s+m] = P[1\dots m]$, instead of comparing each element, compare their hash values

1. What kind of hash functions?
2. How to implement it in time $O(n+m)$?

Implementation

1. Consider $P[1 \dots m]$ as a huge number in radix-26
   $p = P[1] + P[2] \cdot 26 + P[3] \cdot 26^2 + \cdots P[m] \cdot 26^{m-1}$

Implementation

1. Consider $P[1 \dots m]$ as a huge number in radix-26
   $p = P[1] + P[2] \cdot 26 + P[3] \cdot 26^2 + \cdots P[m] \cdot 26^{m-1}$
2. However this number is too large for standard computers — standard machines process integers $< 2^{64}$ in one instruction
3. This is much less than 26^m

Implementation

1. Consider $P[1 \dots m]$ as a huge number in radix-26
   $p = P[1] + P[2] \cdot 26 + P[3] \cdot 26^2 + \cdots P[m] \cdot 26^{m-1}$
2. However this number is too large for standard computers — standard machines process integers $< 2^{64}$ in one instruction
3. This is much less than 26^m
4. Consider hash $h : \mathbb{Z} \to \{0, 1, \dots, q-1\}$ like
   $h[x] = x \mod q$ for $q < 2^{64}$

Implementation

1. Consider $P[1 \dots m]$ as a huge number in radix-26
   $p = P[1] + P[2] \cdot 26 + P[3] \cdot 26^2 + \cdots P[m] \cdot 26^{m-1}$
2. However this number is too large for standard computers — standard machines process integers $< 2^{64}$ in one instruction
3. This is much less than 26^m
4. Consider hash $h : \mathbb{Z} \to \{0, 1, \dots, q-1\}$ like
   $h[x] = x \mod q$ for $q < 2^{64}$
5. So $h[P] := p \mod q$ is string P's hash tag.
6. How to compute hash tags of
   $T[1 \dots m], T[2 \dots m+1], \dots, T[n-m+1 \dots n]$?

1. Let t_i denote the number $T[i+1 \dots i+m]$, i.e., $t_i := T[i+1] + T[i+2] \cdot 26 + T[i+3] \cdot 26^2 + \cdots T[i+m] \cdot 26^{m-1}$

1. Let t_i denote the number $T[i+1 \ldots i+m]$, i.e.,
   $t_i := T[i+1] + T[i+2] \cdot 26 + T[i+3] \cdot 26^2 + \cdots T[i+m] \cdot 26^{m-1}$
2. How to compute $t_0 \mod q, \ldots, t_{n-m} \mod q$ efficiently?

1. Let t_i denote the number $T[i+1 \ldots i+m]$, i.e.,
   $t_i := T[i+1] + T[i+2] \cdot 26 + T[i+3] \cdot 26^2 + \cdots T[i+m] \cdot 26^{m-1}$
2. How to compute $t_0 \mod q, \ldots, t_{n-m} \mod q$ efficiently?
3. OBS: $h(t_i) = \left(h(t_{i+1}) - T[i+1+m] \cdot 26^{m-1} \right) * 26 + T[i+1] \pmod q$ in $O(1)$ time!

1. Let t_i denote the number $T[i+1 \ldots i+m]$, i.e., $t_i := T[i+1] + T[i+2] \cdot 26 + T[i+3] \cdot 26^2 + \cdots T[i+m] \cdot 26^{m-1}$
2. How to compute $t_0 \mod q, \ldots, t_{n-m} \mod q$ efficiently?
3. OBS: $h(t_i) = \left(h(t_{i+1}) - T[i+1+m] \cdot 26^{m-1} \right) * 26 + T[i+1] \pmod q$ in $O(1)$ time!
4. This gives a linear time algorithm to compute hash tags $h(T[1 \ldots m]), \ldots, h(T[n-m+1 \ldots n])$ (assume q if not huge)

Analysis

Question

After calculating the hash tags of $h[P]$ and $h(T[1 \dots m]), \dots, h(T[n-m+1 \dots n])$ in time $O(n+m)$, how to find all shifts s?

Analysis

Question

After calculating the hash tags of $h[P]$ and $h(T[1 \dots m]), \dots, h(T[n-m+1 \dots n])$ in time $O(n+m)$, how to find all shifts s?

1. If $t_s \equiv p \mod q$, output "Matching on shift s!"
2. Question: Why is it correct?

Analysis

Question

After calculating the hash tags of $h[P]$ and $h(T[1\ldots m]),\ldots,h(T[n-m+1\ldots n])$ in time $O(n+m)$, how to find all shifts s?

1. If $t_s \equiv p \mod q$, output "Matching on shift s!"
2. Question: Why is it correct?
3. Case 1: $t_s = p$ — our algorithm will output this shift
4. Case 2: $t_s \neq p$ but $h(t_s) = h(p)$ ☹
5. Choose a large prime q in h, say between $[10n^4, 100n^4]$, and consider $\Pr_q[h(t_s) \equiv h(p) \mod q]$ — How many primes q are there?

1. Our worry is $t_s \neq p$ but $t_s \equiv p \mod q$ — the algorithm outputs a mistake ☹

1. Our worry is $t_s \neq p$ but $t_s \equiv p \mod q$ — the algorithm outputs a mistake ☹
2. This happens only if q divides $|t_s - p|$.

1. Our worry is $t_s \neq p$ but $t_s \equiv p \mod q$ — the algorithm outputs a mistake ☹
2. This happens only if q divides $|t_s - p|$.
3. How large is $|t_s - p|$ and how many prime factors does it have?

1. Our worry is $t_s \neq p$ but $t_s \equiv p \mod q$ — the algorithm outputs a mistake ☹
2. This happens only if q divides $|t_s - p|$.
3. How large is $|t_s - p|$ and how many prime factors does it have?
4. Total number of bad primes is $\leqslant n \cdot \frac{2m}{\log n}$

1. Our worry is $t_s \neq p$ but $t_s \equiv p \mod q$ — the algorithm outputs a mistake ☹
2. This happens only if q divides $|t_s - p|$.
3. How large is $|t_s - p|$ and how many prime factors does it have?
4. Total number of bad primes is $\leqslant n \cdot \frac{2m}{\log n}$
5. None of $t_0, \ldots, t_{n-m}$ gets $h(t_s) = h(p)$ with probability $1 - \frac{2nm/\log n}{20n^4/\log n} \geqslant 1 - \frac{1}{10n^2}$.

Summary

1. If we choose a large prime $q \in [10n^4, 100n^4]$, with.prob. $\geqslant 1 - \frac{1}{10n^2}$, it outputs all correct shifts s without any mistake.
2. Reduce the error probability by taking multiple hash tags

Summary

1. If we choose a large prime $q \in [10n^4, 100n^4]$, with.prob. $\geqslant 1 - \frac{1}{10n^2}$, it outputs all correct shifts s without any mistake.
2. Reduce the error probability by taking multiple hash tags
3. Or compare $T[i+1 \ldots i+m]$ and $P[1 \ldots m]$ when their hash tags are the same — $O(nm)$ in worst case but $O(n+m)$ in good cases

Summary

1. If we choose a large prime $q \in [10n^4, 100n^4]$, with.prob. $\geqslant 1 - \frac{1}{10n^2}$, it outputs all correct shifts s without any mistake.
2. Reduce the error probability by taking multiple hash tags
3. Or compare $T[i+1 \ldots i+m]$ and $P[1 \ldots m]$ when their hash tags are the same — $O(nm)$ in worst case but $O(n+m)$ in good cases
4. However, sampling a prime is non-trivial ☹
5. What if we sample q from $[10n^4, 100n^4]$ without the prime constraint?
6. Next: KMP algorithms ☺

Outline

1. Introduction
2. Rabin-Karp Algorithm
3. KMP Algorithm

Introduction

T= abbaabab   P= abab

s=0: abbaabab / abab
s=1: abbaabab / ab··
s=2: abbaabab / ab··
s=3: abbaabab / abab
s=4: abbaabab / abab

i=1: abbaabab / a
i=2: abbaabab / abab
i=3: abbaabab / abab
i=4: abbaabab / abab
i=5: abbaabab / abab
i=6: abbaabab / abab

Switch the idea

1. For each s, find the longest prefix of P starting from $T[s+1]$
2. For each i, consider the longest prefix of P that ends at $T[i]$

Introduction

T= a b b a a b a b   P= abab

s=0 abbaabab
abab

s=1 abbaabab
ab..

s=2 abbaabab
ab..

s=3 abbaabab
abab

s=4 abbaabab
abab

i=1 abbaabab
a

i=2 abbaabab
abab

i=3 abbaabab
abab

i=4 abbaabab
abab

i=5 abbaabab
abab

i=6 abbaabab
abab

Switch the idea

1. For each s, find the longest prefix of P starting from $T[s+1]$
2. For each i, consider the longest prefix of P that ends at $T[i]$
3. Basic idea: Suppose the longest prefix ends at $T[i]$ is $P[1 \dots q]$, what if $T[i+1] = P[q+1]$?

Introduction

T= abbaabab P=abab

s=0 abbaabab abab

s=1 abbaabab ab··

s=2 abbaabab ab··

s=3 abbaabab abab

s=4 abbaabab abab

i=1 abbaabab a

i=2 abbaabab abab

i=3 abbaabab abab

i=4 abbaabab abab

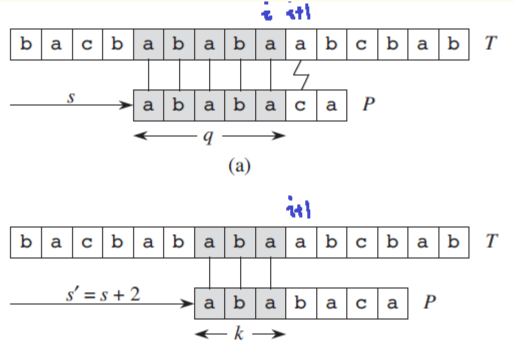
i=5 abbaabab abab

i=6 abbaabab abab

Switch the idea

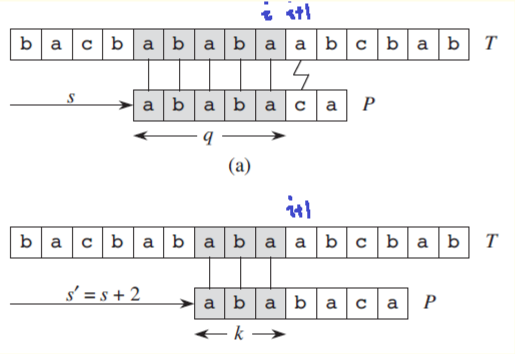
1. For each s, find the longest prefix of P starting from $T[s+1]$
2. For each i, consider the longest prefix of P that ends at $T[i]$
3. Basic idea: Suppose the longest prefix ends at $T[i]$ is $P[1 \ldots q]$, what if $T[i+1] = P[q+1]$?
4. Key question: What shall we do when $T[i+1] \neq P[q+1]$?

When $T[i+1] \neq P[q+1]$



1. If we knew the longest prefix of P (again!) that ends at $P[q]$ is $P[1\ldots k]$

When $T[i+1] \neq P[q+1]$



1. If we knew the longest prefix of P (again!) that ends at $P[q]$ is $P[1 \ldots k]$
2. We can try comparing $T[i+1]$ with $P[k+1]$
3. Reason: $T[i-k+1 \ldots i] = P[1 \ldots k]$ since $T[i-q+1 \ldots i] = P[1 \ldots q]$ and $P[1 \ldots k] = P[q-k+1 \ldots q]$

Implementation

1. Let $\pi[q] = \max\left\{k : k < q \text{ and } P[1 \ldots k] = P[q-k+1 \ldots q]\right\}$ be the longest prefix of P that ends at $P[q]$

Implementation

1. Let $\pi[q] = \max\left\{k : k < q \text{ and } P[1\ldots k] = P[q-k+1\ldots q]\right\}$ be the longest prefix of P that ends at $P[q]$

2. **procedure** KMP-MATCHER

```
procedure KMP-Matcher
    q = 0                                   // longest prefix at i
    for i = 1, ..., n do
        while q > 0 and P[q+1] ≠ T[i] do
            q = π[q]                        // Keeping tracing the prefixes
        if P[q+1] = T[i] then
            q = q + 1
        if q = m then
            output shift at i − m
            q = π[q]                        // look for the next match
```

Implementation

1. Let $\pi[q] = \max\left\{k : k < q \text{ and } P[1\ldots k] = P[q-k+1\ldots q]\right\}$ be the longest prefix of P that ends at $P[q]$

2.
```
procedure KMP-MATCHER
    q = 0                                   // longest prefix at i
    for i = 1, ..., n do
        while q > 0 and P[q+1] ≠ T[i] do
            q = π[q]                        // Keeping tracing the prefixes
        if P[q+1] = T[i] then
            q = q + 1
        if q = m then
            output shift at i − m
            q = π[q]                        // look for the next match
```

3. One more question: How to compute π? — exactly the same problem

```
COMPUTE-PREFIX-FUNCTION(P)
m = P.length
let π[1..m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k + 1] ≠ P[q]
        k = π[k]
    if P[k + 1] == P[q]
        k = k + 1
    π[q] = k
return π
```

```
procedure KMP-MATCHER
    q = 0                                   // longest prefix at i
    for i = 1, . . . , n do
        while q > 0 and P[q + 1] ≠ T[i] do
            q = π[q]                        // Keeping tracing the prefixes
        if P[q + 1] = T[i] then
            q = q + 1
        if q = m then
            output shift at i − m
            q = π[q]                        // look for the next match
```

Analysis

Running Time

$O(n+m)$: Apply amortized analysis and control the while loop by a potential function

Analysis

Running Time

$O(n+m)$: Apply amortized analysis and control the while loop by a potential function

Correctness: Basic properties of π

Apply induction on q to show $\pi[q]$ is correct in our ALGO:

Analysis

Running Time

$O(n+m)$: Apply amortized analysis and control the while loop by a potential function

Correctness: Basic properties of π

Apply induction on q to show $\pi[q]$ is correct in our ALGO:

1. $\pi^*[q-1] \subset \{1, 2, \ldots, q-2\}$ contains all prefixes of P ending at $q-1$
2. Lemma 32.5 in CLRS: If $\pi[1], \ldots, \pi[q-1]$ are correct, $\pi^*[q-1] = \{\pi[q-1], \pi[\pi[q-1]], \ldots, \pi^{(t)}[q-1], \ldots\}$

Analysis

Running Time

$O(n+m)$: Apply amortized analysis and control the while loop by a potential function

Correctness: Basic properties of π

Apply induction on q to show $\pi[q]$ is correct in our ALGO:

1. $\pi^*[q-1] \subset \{1, 2, \ldots, q-2\}$ contains all prefixes of P ending at $q-1$
2. Lemma 32.5 in CLRS: If $\pi[1], \ldots, \pi[q-1]$ are correct, $\pi^*[q-1] = \{\pi[q-1], \pi[\pi[q-1]], \ldots, \pi^{(t)}[q-1], \ldots\}$
3. Now consider $\pi[q]$ *vs* $\pi^*[q-1]$
4. Lemma 32.6 in CLRS: $\pi[q] > 0$ implies that $\pi[q] - 1 \in \pi^*[q-1]$

Analysis

Running Time

$O(n+m)$: Apply amortized analysis and control the while loop by a potential function

Correctness: Basic properties of π

Apply induction on q to show $\pi[q]$ is correct in our ALGO:

1. $\pi^*[q-1] \subset \{1, 2, \ldots, q-2\}$ contains all prefixes of P ending at $q-1$
2. Lemma 32.5 in CLRS: If $\pi[1], \ldots, \pi[q-1]$ are correct, $\pi^*[q-1] = \{\pi[q-1], \pi[\pi[q-1]], \ldots, \pi^{(t)}[q-1], \ldots\}$
3. Now consider $\pi[q]$ *vs* $\pi^*[q-1]$
4. Lemma 32.6 in CLRS: $\pi[q] > 0$ implies that $\pi[q] - 1 \in \pi^*[q-1]$
5. Corollary 32.7 in CLRS: $\pi[q] = \max \left\{ k \in \pi^*[q-1] : P[k+1] = P[q] \right\}$

Analysis

Running Time

$O(n+m)$: Apply amortized analysis and control the while loop by a potential function

Correctness: Basic properties of π

Apply induction on q to show $\pi[q]$ is correct in our ALGO:

1. $\pi^*[q-1] \subset \{1, 2, \ldots, q-2\}$ contains all prefixes of P ending at $q-1$
2. Lemma 32.5 in CLRS: If $\pi[1], \ldots, \pi[q-1]$ are correct, $\pi^*[q-1] = \{\pi[q-1], \pi[\pi[q-1]], \ldots, \pi^{(t)}[q-1], \ldots\}$
3. Now consider $\pi[q]$ *vs* $\pi^*[q-1]$
4. Lemma 32.6 in CLRS: $\pi[q] > 0$ implies that $\pi[q] - 1 \in \pi^*[q-1]$
5. Corollary 32.7 in CLRS: $\pi[q] = \max\left\{k \in \pi^*[q-1] : P[k+1] = P[q]\right\}$

Extending the above proof shows the correctness of KMP

Summary

1. Miller-Rabin Algorithm: Easy to implement but relies on random prime numbers
2. KMP Algorithm: Tricky to design but fast and reliable
3. Automata, data structures for suffixes and prefixes, ...

Questions?

```
COMPUTE-PREFIX-FUNCTION(P)
m = P.length
let π[1..m] be a new array
π[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k + 1] ≠ P[q]
        k = π[k]
    if P[k + 1] == P[q]
        k = k + 1
    π[q] = k
return π
```

```
procedure EULER-SIEVE(n)
    Set h[2, ..., n] = True                    // h[q] indicates q is a prime or not
    Set list_p = ∅                              // list_p is the list of primes
    for i = 2, ..., n do
        if h[i] = True then
            Add i to list_p                     // i is a prime
        for each p ∈ list_p do
            h[p ∗ i] = False                    // p ∗ i is not a prime
            if i mod p = 0 then
                Break the for-loop of p;
```