Efficient Sequential Algorithms, Comp309

University of Liverpool

2010–2011

Module Organiser, Igor Potapov

Part 2: Pattern Matching

References: T. H. Cormen, C. E. Leiserson, R. L. Rivest **Introduction to Algorithms**, Second Edition. MIT Press (2001). "String Matching"

Aho, Alfred V, *Algorithms for finding patterns in Strings*, Handbook of Theoretical Computer Science, Edited by J van Leeuwen, Elsevier (1990), 255–300.

String Matching

Finding all occurrences of a pattern in a text is a problem that arises frequently in various contexts:

text editing typically the text is a document being edited, and the pattern searched for is a particular word supplied by the user.

DNA mapping in this case we are interested in finding a particular pattern in a (long) DNA sequence.

WWW searching the text is the union of all web-pages in the internet

String Matching

Finding all occurrences of a pattern in a text is a problem that arises frequently in various contexts:

text editing typically the text is a document being edited, and the pattern searched for is a particular word supplied by the user.

DNA mapping in this case we are interested in finding a particular pattern in a (long) DNA sequence.

WWW searching the text is the union of all web-pages in the internet

String Matching

Finding all occurrences of a pattern in a text is a problem that arises frequently in various contexts:

text editing typically the text is a document being edited, and the pattern searched for is a particular word supplied by the user.

DNA mapping in this case we are interested in finding a particular pattern in a (long) DNA sequence.

WWW searching the text is the union of all web-pages in the internet

String Matching

Finding all occurrences of a pattern in a text is a problem that arises frequently in various contexts:

text editing typically the text is a document being edited, and the pattern searched for is a particular word supplied by the user.

DNA mapping in this case we are interested in finding a particular pattern in a (long) DNA sequence.

WWW searching the text is the union of all web-pages in the internet

Problem definition.

Given an alphabet $\mathcal{A}$, a text T (an array of n characters in $\mathcal{A}$) and a pattern P (another array of $m \leq n$ characters in $\mathcal{A}$), we say that P *occurs with shift* s *in* T (or P *occurs beginning at position* $s+1$ *in* T) if $0 \leq s \leq n-m$ and $T[s+j] = P[j]$ for $1 \leq j \leq m$. A shift is *valid* if P occurs with shift s in T and *invalid* otherwise. The string-matching problem is the problem of finding all valid shifts for a given choice of P and T.

Example

$$T \equiv \text{tadadattaetadadadafa}$$

$$P \equiv \text{dada}$$

Valid shifts are two, twelve and fourteen.

Problem definition.

Given an alphabet $\mathcal{A}$, a text T (an array of n characters in $\mathcal{A}$) and a pattern P (another array of $m \leq n$ characters in $\mathcal{A}$), we say that *P occurs with shift s in T* (or *P occurs beginning at position $s+1$ in T*) if $0 \leq s \leq n-m$ and $T[s+j] = P[j]$ for $1 \leq j \leq m$. A shift is *valid* if P occurs with shift s in T and *invalid* otherwise. The string-matching problem is the problem of finding all valid shifts for a given choice of P and T.

Example

$$T \equiv \text{tadadattaetadadadafa}$$

$$P \equiv \text{dada}$$

Valid shifts are two, twelve and fourteen.

Problem definition.

Given an alphabet $\mathcal{A}$, a text T (an array of n characters in $\mathcal{A}$) and a pattern P (another array of $m \leq n$ characters in $\mathcal{A}$), we say that P *occurs with shift* s *in* T (or P *occurs beginning at position* $s+1$ *in* T) if $0 \leq s \leq n-m$ and $T[s+j] = P[j]$ for $1 \leq j \leq m$. A shift is *valid* if P occurs with shift s in T and *invalid* otherwise. The string-matching problem is the problem of finding all valid shifts for a given choice of P and T.

Example

$$T \equiv \text{tadadattaetadadadafa}$$

$$P \equiv \text{dada}$$

Valid shifts are two, twelve and fourteen.

Problem definition.

Given an alphabet $\mathcal{A}$, a text T (an array of n characters in $\mathcal{A}$) and a pattern P (another array of $m \leq n$ characters in $\mathcal{A}$), we say that *P occurs with shift s in T* (or *P occurs beginning at position $s+1$ in T*) if $0 \leq s \leq n-m$ and $T[s+j] = P[j]$ for $1 \leq j \leq m$. A shift is *valid* if P occurs with shift s in T and *invalid* otherwise. The string-matching problem is the problem of finding all valid shifts for a given choice of P and T.

Example

$$T \equiv \text{tadadattaetadadadafa}$$

$$P \equiv \text{dada}$$

Valid shifts are two, twelve and fourteen.

Problem definition.

Given an alphabet $\mathcal{A}$, a text T (an array of n characters in $\mathcal{A}$) and a pattern P (another array of $m \leq n$ characters in $\mathcal{A}$), we say that P *occurs with shift* s *in* T (or P *occurs beginning at position* $s+1$ *in* T) if $0 \leq s \leq n-m$ and $T[s+j] = P[j]$ for $1 \leq j \leq m$. A shift is *valid* if P occurs with shift s in T and *invalid* otherwise. The string-matching problem is the problem of finding all valid shifts for a given choice of P and T.

Example

$$T \equiv \text{tadadattaetadadadafa}$$

$$P \equiv \text{dada}$$

Valid shifts are two, twelve and fourteen.

Problem definition.

Given an alphabet $\mathcal{A}$, a text T (an array of n characters in $\mathcal{A}$) and a pattern P (another array of $m \leq n$ characters in $\mathcal{A}$), we say that *P occurs with shift s in T* (or *P occurs beginning at position $s+1$ in T*) if $0 \leq s \leq n-m$ and $T[s+j] = P[j]$ for $1 \leq j \leq m$. A shift is *valid* if P occurs with shift s in T and *invalid* otherwise. The string-matching problem is the problem of finding all valid shifts for a given choice of P and T.

Example

$$T \equiv \text{tadadattaetadadadafa}$$

$$P \equiv \text{dada}$$

Valid shifts are two, twelve and fourteen.

A Short History

First came the obvious brute-force algorithm (still in widespread use).

Its worst-case running time is $O(nm)$. According to Aho, the expected performance is usually $O(m+n)$ "in practical situations."

A Short History

First came the obvious brute-force algorithm (still in widespread use).

Its worst-case running time is $O(nm)$. According to Aho, the expected performance is usually $O(m+n)$ "in practical situations."

In 1970, Cook proved a theoretical result showing that a “2-way deterministic pushdown automaton” language can be recognized in linear time on a random-access machine. This result implies that there is an algorithm which solves the pattern-matching problem in time $O(n + m)$ (worst case), because the language

$$\{P\#T \mid T = xPy \text{ for some } x \text{ and } y\}$$

can be recognized by a 2DPDA.

His theorem did not explicitly provide the algorithm.

Knuth traced the simulation from Cook's proof to derive a linear-time pattern matching algorithm. Pratt modified the algorithm to make its running time independent of alphabet size.

This algorithm was independently discovered by J.H. Morris, who was implementing a text editor, and wanted to avoid "backing up" in the text string.

Knuth, Morris, and Pratt didn't get around to publishing their algorithm until 1976, and in the meantime R. S. Boyer and J. S. Moore (and, independently, R. W. Gosper) discovered a different algorithm which is much faster in many applications, since it often examines only a fraction of the characters in the text string. Many text editors use this algorithm.

Knuth traced the simulation from Cook's proof to derive a linear-time pattern matching algorithm. Pratt modified the algorithm to make its running time independent of alphabet size.

This algorithm was independently discovered by J.H. Morris, who was implementing a text editor, and wanted to avoid "backing up" in the text string.

Knuth, Morris, and Pratt didn't get around to publishing their algorithm until 1976, and in the meantime R. S. Boyer and J. S. Moore (and, independently, R. W. Gosper) discovered a different algorithm which is much faster in many applications, since it often examines only a fraction of the characters in the text string. Many text editors use this algorithm.

Knuth traced the simulation from Cook's proof to derive a linear-time pattern matching algorithm. Pratt modified the algorithm to make its running time independent of alphabet size.

This algorithm was independently discovered by J.H. Morris, who was implementing a text editor, and wanted to avoid "backing up" in the text string.

Knuth, Morris, and Pratt didn't get around to publishing their algorithm until 1976, and in the meantime R. S. Boyer and J. S. Moore (and, independently, R. W. Gosper) discovered a different algorithm which is much faster in many applications, since it often examines only a fraction of the characters in the text string. Many text editors use this algorithm.

If you look on the web, you can find comparisons of these algorithms, and also some nice animations.

Notation (1)

$|s|$ if s is a string, denotes the *length* of s, i.e. the number of characters in the string.

$\mathcal{A}^*$ ("A-star") the set of all finite-length strings formed using characters from the alphabet $\mathcal{A}$.

ε the *empty string*, the unique string of length 0.

st is the *concatenation* of strings s and t, obtained by appending the characters of t after those of s. Clearly $|st| = |s| + |t|$.

Notation (1)

$|s|$ if s is a string, denotes the *length* of s, i.e. the number of characters in the string.

$\mathcal{A}^*$ ("A-star") the set of all finite-length strings formed using characters from the alphabet $\mathcal{A}$.

ε the *empty string*, the unique string of length 0.

st is the *concatenation* of strings s and t, obtained by appending the characters of t after those of s. Clearly $|st| = |s| + |t|$.

Notation (1)

$|s|$ if s is a string, denotes the *length* of s, i.e. the number of characters in the string.

$\mathcal{A}^*$ ("A-star") the set of all finite-length strings formed using characters from the alphabet $\mathcal{A}$.

ε the *empty string*, the unique string of length 0.

st is the *concatenation* of strings s and t, obtained by appending the characters of t after those of s. Clearly $|st| = |s| + |t|$.

Notation (1)

$|s|$ if s is a string, denotes the *length* of s, i.e. the number of characters in the string.

$\mathcal{A}^*$ ("A-star") the set of all finite-length strings formed using characters from the alphabet $\mathcal{A}$.

ε the *empty string*, the unique string of length 0.

st is the *concatenation* of strings s and t, obtained by appending the characters of t after those of s. Clearly $|st| = |s| + |t|$.

Notation (2)

- A string w is a *prefix* of x, if $x \equiv wy$ for some string $y \in \mathcal{A}^*$. Of course the length of w cannot be larger than that of x.
- A string w is a *suffix* of x, if $x = yw$ for some string $y \in \mathcal{A}^*$. We have $|w| \leq |x|$.
- If x, y are both suffixes of z then only three cases arise: if $|x| \leq |y|$ then x is also a suffix of y. Otherwise, y is a suffix of x. If particular $|x| = |y|$ implies $x = y$.

Notation (2)

- A string w is a *prefix* of x, if $x \equiv wy$ for some string $y \in \mathcal{A}^*$. Of course the length of w cannot be larger than that of x.
- A string w is a *suffix* of x, if $x = yw$ for some string $y \in \mathcal{A}^*$. We have $|w| \leq |x|$.
- If x, y are both suffixes of z then only three cases arise: if $|x| \leq |y|$ then x is also a suffix of y. Otherwise, y is a suffix of x. If particular $|x| = |y|$ implies $x = y$.

Notation (2)

- A string w is a *prefix* of x, if $x \equiv wy$ for some string $y \in \mathcal{A}^*$. Of course the length of w cannot be larger than that of x.
- A string w is a *suffix* of x, if $x = yw$ for some string $y \in \mathcal{A}^*$. We have $|w| \leq |x|$.
- If x, y are both suffixes of z then only three cases arise: if $|x| \leq |y|$ then x is also a suffix of y. Otherwise, y is a suffix of x. If particular $|x| = |y|$ implies $x = y$.

Notation (3)

- We shall denote the k character prefix of a pattern, say, P, by P_k. Thus $P_0 = \varepsilon$, $P_m = P$, and the pattern matching problem is that of finding all shifts s such that P is a suffix of T_{s+m}.
- We will assume that checking equality between two strings takes time proportional to the length of the shortest of the two.

Notation (3)

- We shall denote the k character prefix of a pattern, say, P, by P_k. Thus $P_0 = \varepsilon$, $P_m = P$, and the pattern matching problem is that of finding all shifts s such that P is a suffix of T_{s+m}.
- We will assume that checking equality between two strings takes time proportional to the length of the shortest of the two.

Brute-Force

The obvious method that immediately comes to mind is just to check, for each possible position in the text, whether the pattern does in fact match the text.

```
BRUTE-MATCHING (T, P)
    n ← length(T)
    m ← length(P)
    for s ← 0 to n − m
        if P = T[s + 1..s + m]
            print "pattern occurs with shift s"
```

How good is BRUTE-MATCHING

The algorithm finds all valid shifts in time $\Theta((n-m+1)m)$.

It is easy to verify the upper bound by inspecting the code.

If $T = aa\ldots a$ (n times) and P is a substring of length m of T, the algorithm BRUTE-MATCHING will actually spend $O(m)$ time for each of the possible $n-m+1$ positions.

How good is BRUTE-MATCHING

The algorithm finds all valid shifts in time $\Theta((n-m+1)m)$.

It is easy to verify the upper bound by inspecting the code.

If $T = aa\ldots a$ (n times) and P is a substring of length m of T, the algorithm BRUTE-MATCHING will actually spend $O(m)$ time for each of the possible $n-m+1$ positions.

How good is BRUTE-MATCHING

The algorithm finds all valid shifts in time $\Theta((n-m+1)m)$.

It is easy to verify the upper bound by inspecting the code.

If $T = aa\ldots a$ (n times) and P is a substring of length m of T, the algorithm BRUTE-MATCHING will actually spend $O(m)$ time for each of the possible $n-m+1$ positions.

Examples

Consider the following text:

$$0101010101010100101010101010101010100101010101$$

and the pattern

$$010100$$

The algorithm is often very good in practice. Find the word joy in the following text:

> *A very popular definition of Argumentation is the one given in an important Handbook, which in a way testifies the current period of good fortune which Argumentation Theory is enjoying. The definition characterises the process of argumentation as a "verbal and social activity of reason aimed at increasing (or decreasing) the acceptability of a controversial standpoint for the listener, by putting forward a constellation of proposition intended to justify (or refute) the standpoint before a rational judge". Having described many of the characteristics of argumentation theory in the previous chapter, here we will concentrate on an important aspect of the arguing activity, that is that it is a process involving two entities: a listener and a speaker. Dialogue, therefore, is paramount to argumentation. The present chapter will examine the dialogic component of the argumentation activities, by providing a survey of literature in this subject, and concluding with our own approach to the treatment of this aspect.*

The text contains only 4 substrings matching j, and just a single substring matching both jo and joy:

> *A very popular definition of Argumentation is the one given in an important Handbook, which in a way testifies the current period of good fortune which Argumentation Theory is en j oying. The definition characterises the process of argumentation as a "verbal and social activity of reason aimed at increasing (or decreasing) the acceptability of a controversial standpoint for the listener, by putting forward a constellation of proposition intended to j ustify (or refute) the standpoint before a rational j udge". Having described many of the characteristics of argumentation theory in the previous chapter, here we will concentrate on an important aspect of the arguing activity, that is that it is a process involving two entities: a listener and a speaker. Dialogue, therefore, is paramount to argumentation. The present chapter will examine the dialogic component of the argumentation activities, by providing a survey of literature in this sub j ect, and concluding with our own approach to the treatment of this aspect.*

What is wrong with BRUTE-MATCHING?

The inefficiency comes from not making proper use of the partial matches.

If a pattern like: 1010010100110111 has been matched to 10100101001... (and then a miss-match occurs) we may argue that we don't need to reset the pointer to the T string to consider 1 0100101001... .

The partial match (and our knowledge of the pattern) tells us that, at least, we may restart our comparison from 10100 101001... . The subsequent quest for a $O(n+m)$ algorithm focused on efficient ways of using this information.

What is wrong with BRUTE-MATCHING?

The inefficiency comes from not making proper use of the partial matches.

If a pattern like: 1010010100110111 has been matched to 10100101001... (and then a miss-match occurs) we may argue that we don't need to reset the pointer to the T string to consider 1 0100101001... .

The partial match (and our knowledge of the pattern) tells us that, at least, we may restart our comparison from 10100 101001... . The subsequent quest for a $O(n+m)$ algorithm focused on efficient ways of using this information.

String matching with finite automata

An alternative approach that, nearly, overcomes the inefficiencies of BRUTE-MATCHING is based on the idea of avoiding any backward movements on the text T by keeping information about portions of T that partially match P. The amount of information that is kept about these partial matches depends on the size of the pattern but not on the size of the text.

The most natural framework to describe such information is provided by the Theory of Finite State machines (or Automata).

The construction of the automaton is based on the pattern P and the automaton is used to find matches (and partial matches) of P in the text T.

String matching with finite automata

An alternative approach that, nearly, overcomes the inefficiencies of BRUTE-MATCHING is based on the idea of avoiding any backward movements on the text T by keeping information about portions of T that partially match P. The amount of information that is kept about these partial matches depends on the size of the pattern but not on the size of the text.

The most natural framework to describe such information is provided by the Theory of Finite State machines (or Automata).

The construction of the automaton is based on the pattern P and the automaton is used to find matches (and partial matches) of P in the text T.

String matching with finite automata

An alternative approach that, nearly, overcomes the inefficiencies of BRUTE-MATCHING is based on the idea of avoiding any backward movements on the text T by keeping information about portions of T that partially match P. The amount of information that is kept about these partial matches depends on the size of the pattern but not on the size of the text.

The most natural framework to describe such information is provided by the Theory of Finite State machines (or Automata).

The construction of the automaton is based on the pattern P and the automaton is used to find matches (and partial matches) of P in the text T.

Finite State Machines

A (deterministic) *finite state machine* or *automaton M* is defined by

$\mathcal{A}$ a finite alphabet;

Q, a finite set of *states*: $Q = \{q_0, q_1, q_2, \ldots, q_k\}$;

$S \in Q$, the *initial* state;

$F \subseteq Q$, the set of *final* states;

$\delta : Q \times \mathcal{A} \to Q$, the *state-transition function*.

Given string in $\mathcal{A}^*$, the automaton will start in state S. It will then read one character at a time. Each time it reads a character, it uses the transition function to decide whether to move from one state to another.

Finite State Machines: real-life examples

A CD player controller.

A lift.

A graphical user interface menu structure.

Automata as acceptors

A *language* L is a set of strings in $\mathcal{A}^*$.

An automaton *accepts* L if for each string w, M, started in the initial state on the leftmost character of w, after scanning all characters of w, enters a final state if and only if $w \in L$.

Automata as acceptors

A *language* L is a set of strings in $\mathcal{A}^*$.

An automaton *accepts* L if for each string w, M, started in the initial state on the leftmost character of w, after scanning all characters of w, enters a final state if and only if $w \in L$.

Example

$\mathcal{A} = \{0, 1, \ldots, 9\}$

$Q = \{\text{BEGIN, EVEN, ODD}\}$

$S = \text{BEGIN}$

$F = \{\text{EVEN}\}$

Here's the definition of the function δ.

	$s \in \mathcal{A}$									
q	0	1	2	3	4	5	6	7	8	9
BEGIN	EVEN	ODD	EVEN	ODD	EVEN	ODD	EVEN	ODD	EVEN	ODD
EVEN	EVEN	ODD	EVEN	ODD	EVEN	ODD	EVEN	ODD	EVEN	ODD
ODD	EVEN	ODD	EVEN	ODD	EVEN	ODD	EVEN	ODD	EVEN	ODD

A quick simulation

On reading 125, the automaton will start in the state BEGIN, reading the leftmost digit (that's "1"). $\delta(\text{BEGIN},1) = \text{ODD}$, so the automaton moves to state "ODD", before reading the next digit. Now it reads "2", since $\delta(\text{ODD}, 2) = \text{EVEN}$, the automaton moves to state "EVEN". Then reads "5" and, since $\delta(\text{EVEN}, 5) = \text{ODD}$, the automaton moves to state "ODD". Since the number is finished "ODD" is the final answer of the automaton!

Exercise. Write an automaton that recognises multiples of 5. Write an automaton that recognizes multiples of 3.

Trick: a number is a multiple of 3 iff the sum of its digits is a multiple of 3.

A quick simulation

On reading 125, the automaton will start in the state BEGIN, reading the leftmost digit (that's "1"). $\delta(\text{BEGIN},1) = \text{ODD}$, so the automaton moves to state "ODD", before reading the next digit. Now it reads "2", since $\delta(\text{ODD}, 2) = \text{EVEN}$, the automaton moves to state "EVEN". Then reads "5" and, since $\delta(\text{EVEN}, 5) = \text{ODD}$, the automaton moves to state "ODD". Since the number is finished "ODD" is the final answer of the automaton!

Exercise. Write an automaton that recognises multiples of 5. Write an automaton that recognizes multiples of 3.

Trick: a number is a multiple of 3 iff the sum of its digits is a multiple of 3.

A quick simulation

On reading 125, the automaton will start in the state BEGIN, reading the leftmost digit (that's "1"). $\delta(\text{BEGIN},1) = \text{ODD}$, so the automaton moves to state "ODD", before reading the next digit. Now it reads "2", since $\delta(\text{ODD}, 2) = \text{EVEN}$, the automaton moves to state "EVEN". Then reads "5" and, since $\delta(\text{EVEN}, 5) = \text{ODD}$, the automaton moves to state "ODD". Since the number is finished "ODD" is the final answer of the automaton!

Exercise. Write an automaton that recognises multiples of 5. Write an automaton that recognizes multiples of 3.

Trick: a number is a multiple of 3 iff the sum of its digits is a multiple of 3.

String-matching automaton.

There will be an automaton for each pattern P. First an important definition:

bookmark sigma def!

Given a pattern P, let σ be a function that, for any string x, returns the length of the longest prefix of P that is a suffix of x.

Examples. Let $P \equiv$abc. Then $\sigma(\text{caba}) = 1$, $\sigma(\text{cabab}) = 2$, and $\sigma(\text{cababc}) = 3$.

String-matching automaton.

There will be an automaton for each pattern P. First an important definition:

bookmark sigma def!

> Given a pattern P, let σ be a function that, for any string x, returns the length of the longest prefix of P that is a suffix of x.

Examples. Let $P \equiv$abc. Then $\sigma(\text{caba}) = 1$, $\sigma(\text{cabab}) = 2$, and $\sigma(\text{cababc}) = 3$.

String-matching automaton.

There will be an automaton for each pattern P. First an important definition:

bookmark sigma def!

> Given a pattern P, let σ be a function that, for any string x, returns the length of the longest prefix of P that is a suffix of x.

Examples. Let $P \equiv$abc. Then $\sigma(\text{caba}) = 1$, $\sigma(\text{cabab}) = 2$, and $\sigma(\text{cababc}) = 3$.

Motivation: what will the algorithm do?

```
for i ← 1 to n
    q ← σ(T_i)
    if q = m
        print "pattern occurs with shift i − m"
            (match ends at position i)
```

Idea: Say $q = \sigma(T_i)$ and $T_{i+1} = T_i a$. We will show that $\sigma(T_{i+1})$ (the new q) is $\sigma(P_q a)$ — it just depends on P and a.

Motivation: what will the algorithm do?

```
for i ← 1 to n
    q ← σ(T_i)
    if q = m
        print "pattern occurs with shift i − m"
            (match ends at position i)
```

Idea: Say $q = \sigma(T_i)$ and $T_{i+1} = T_i a$. We will show that $\sigma(T_{i+1})$ (the new q) is $\sigma(P_q a)$ — it just depends on P and a.

Here's the definition of the automaton!

- $Q = \{0, 1, \ldots, m\}$.
- $q_0 = 0$
- The only accepting state is m.
- The input alphabet is $\mathcal{A}$.
- The transition function is defined by the following equation:

$$\delta(q, x) = \sigma(P_q x)$$

How is this to be used?

```
FINITE-AUTOMATON-MATCHING (T, δ, m)
    n ← length(T)
    q ← 0
    for i ← 1 to n
        q ← δ(q, T[i])
        if q = m
            print "pattern occurs with shift i − m"
```

bookmark automaton alg!

Example

Say the alphabet is $\{a, b, c\}$ and the pattern P is abc. Let's define the automaton.

- $Q = \{0, 1, 2, 3\}$ — the automaton has four states.
- The initial state is (always) $q_0 = 0$.
- The only accepting state is 3.

Example

Say the alphabet is $\{a, b, c\}$ and the pattern P is abc. Let's define the automaton.

- $Q = \{0, 1, 2, 3\}$ — the automaton has four states.
- The initial state is (always) $q_0 = 0$.
- The only accepting state is 3.

Transition function

Recall $P = \text{abc}$. We can represent the transition function as a table with rows indexed by characters and columns indexed by states.

Let's start with $\delta(0,\text{a})$. By definition of σ this is "the length of the longest prefix of P that is a suffix of $P_0\text{a}(\equiv \text{a})$". Therefore $\delta(0,\text{a}) = 1$. Next comes $\delta(1,\text{a})$. This is "the length of the longest prefix of P that is a suffix of $P_1\text{a}(\equiv \text{aa})$". Again $\delta(1,\text{a}) = 1$. Iterating this process one can get δ's full definition (reported in the following table).

	0	1	2	3
a	1	1	1	1
b	0	2	0	0
c	0	0	3	0

Simulation

Let $T =$ aababcabcbb.

T		a	a	b	a	b	c	a	b	c	b	b
i		1	2	3	4	5	6	7	8	9	10	11
q	0	1	1	2	1	2	3	1	2	3	0	0
output							3			6		

FINITE-AUTOMATON-MATCHING (T, δ, m)

$n \leftarrow$ length(T)
$q \leftarrow 0$
for $i \leftarrow 1$ **to** n
  $q \leftarrow \delta(q, T[i])$
  if $q = m$
    print "pattern occurs with shift $i - m$"

Exercise: Another simulation

Simulate the string-matching automaton algorithm for

$$P \equiv \text{abababa}$$

and

$$T = \text{abacbababababaacbacaababababaababababababacac.}$$

Hints. You will need to:

1. Define the automaton.
2. Simulate the algorithm FINITE-AUTOMATON-MATCHING step by step.

Time complexity analysis ... cheating!

- The simple loop structure of FINITE-AUTOMATON-MATCHING implies that its running time is $O(|T|)$.
- However, this does not include the time to compute the transition function δ: we will look at this later!
- Correctness?
  Let's start by understanding what correctness means.

Time complexity analysis ... cheating!

- The simple loop structure of FINITE-AUTOMATON-MATCHING implies that its running time is $O(|T|)$.
- However, this does not include the time to compute the transition function δ: we will look at this later!
- Correctness?
  Let's start by understanding what correctness means.

Time complexity analysis ... cheating!

- The simple loop structure of FINITE-AUTOMATON-MATCHING implies that its running time is $O(|T|)$.
- However, this does not include the time to compute the transition function δ: we will look at this later!
- Correctness?

  Let's start by understanding what correctness means.

Time complexity analysis ... cheating!

- The simple loop structure of FINITE-AUTOMATON-MATCHING implies that its running time is $O(|T|)$.
- However, this does not include the time to compute the transition function δ: we will look at this later!
- Correctness?
  Let's start by understanding what correctness means.

Main Result

Main Result: For each $i \leq n$, the value of q after the ith iteration of the main for loop in FINITE-AUTOMATON-MATCHING is $\sigma(T_i)$, i.e. the length of the longest prefix of the pattern P that is a suffix of T_i.

By definition of σ, $\sigma(T_i) = m$ iff P is a suffix of T_i. Thus, the main result implies that the algorithm returns all valid shifts.

Main Result

Main Result: For each $i \leq n$, the value of q after the ith iteration of the main for loop in FINITE-AUTOMATON-MATCHING is $\sigma(T_i)$, i.e. the length of the longest prefix of the pattern P that is a suffix of T_i.

By definition of σ, $\sigma(T_i) = m$ iff P is a suffix of T_i. Thus, the main result implies that the algorithm returns all valid shifts.

Main Result: For each $i \leq n$, the value of q after the ith iteration of the main for loop in FINITE-AUTOMATON-MATCHING is $\sigma(T_i)$, i.e. the length of the longest prefix of the pattern P that is a suffix of T_i.

We will prove the main result by induction on i.

If $i = 0$, then $T_0 = \varepsilon$ and the theorem holds.

Main Result: For each $i \leq n$, the value of q after the ith iteration of the main for loop in FINITE-AUTOMATON-MATCHING is $\sigma(T_i)$, i.e. the length of the longest prefix of the pattern P that is a suffix of T_i.

We will prove the main result by induction on i.

If $i = 0$, then $T_0 = \varepsilon$ and the theorem holds.

Main Result: For each $i \leq n$, the value of q after the ith iteration of the main for loop in FINITE-AUTOMATON-MATCHING is $\sigma(T_i)$, i.e. the length of the longest prefix of the pattern P that is a suffix of T_i.

For the inductive step, suppose $q = \sigma(T_i)$ after the i'th iteration.

In the $(i+1)$st iteration, the new value of q is assigned to be $\delta(q, T[i+1])$.

We want to show that this is equal to $\sigma(T_{i+1})$.

Main Result: For each $i \leq n$, the value of q after the ith iteration of the main for loop in FINITE-AUTOMATON-MATCHING is $\sigma(T_i)$, i.e. the length of the longest prefix of the pattern P that is a suffix of T_i.

For the inductive step, suppose $q = \sigma(T_i)$ after the i'th iteration.

In the $(i+1)$st iteration, the new value of q is assigned to be $\delta(q, T[i+1])$.

We want to show that this is equal to $\sigma(T_{i+1})$.

Main Result: For each $i \leq n$, the value of q after the ith iteration of the main for loop in FINITE-AUTOMATON-MATCHING is $\sigma(T_i)$, i.e. the length of the longest prefix of the pattern P that is a suffix of T_i.

For the inductive step, suppose $q = \sigma(T_i)$ after the i'th iteration.

In the $(i+1)$st iteration, the new value of q is assigned to be $\delta(q, T[i+1])$.

We want to show that this is equal to $\sigma(T_{i+1})$.

First, a little lemma.

Lemma: Suppose $\sigma(T_i) = q$. Then $\sigma(T_{i+1}) \leq q + 1$.

Proof:

Suppose for contradiction that $\sigma(T_{i+1}) = r > q + 1$.

Then P_r is a suffix of T_{i+1}.

So P_{r-1} is a suffix of T_i.

So $\sigma(T_i) \geq r - 1 > q$.

First, a little lemma.

Lemma: Suppose $\sigma(T_i) = q$. Then $\sigma(T_{i+1}) \leq q + 1$.

Proof:

Suppose for contradiction that $\sigma(T_{i+1}) = r > q + 1$.

Then P_r is a suffix of T_{i+1}.

So P_{r-1} is a suffix of T_i.

So $\sigma(T_i) \geq r - 1 > q$.

First, a little lemma.

Lemma: Suppose $\sigma(T_i) = q$. Then $\sigma(T_{i+1}) \leq q + 1$.

Proof:

Suppose for contradiction that $\sigma(T_{i+1}) = r > q + 1$.

Then P_r is a suffix of T_{i+1}.

So P_{r-1} is a suffix of T_i.

So $\sigma(T_i) \geq r - 1 > q$.

First, a little lemma.

Lemma: Suppose $\sigma(T_i) = q$. Then $\sigma(T_{i+1}) \leq q + 1$.

Proof:

Suppose for contradiction that $\sigma(T_{i+1}) = r > q + 1$.

Then P_r is a suffix of T_{i+1}.

So P_{r-1} is a suffix of T_i.

So $\sigma(T_i) \geq r - 1 > q$.

First, a little lemma.

Lemma: Suppose $\sigma(T_i) = q$. Then $\sigma(T_{i+1}) \leq q + 1$.

Proof:

Suppose for contradiction that $\sigma(T_{i+1}) = r > q + 1$.

Then P_r is a suffix of T_{i+1}.

So P_{r-1} is a suffix of T_i.

So $\sigma(T_i) \geq r - 1 > q$.

First, a little lemma.

Lemma: Suppose $\sigma(T_i) = q$. Then $\sigma(T_{i+1}) \leq q + 1$.

Proof:

Suppose for contradiction that $\sigma(T_{i+1}) = r > q + 1$.

Then P_r is a suffix of T_{i+1}.

So P_{r-1} is a suffix of T_i.

So $\sigma(T_i) \geq r - 1 > q$.

First, a little lemma.

Lemma: Suppose $\sigma(T_i) = q$. Then $\sigma(T_{i+1}) \leq q + 1$.

Proof:

Suppose for contradiction that $\sigma(T_{i+1}) = r > q + 1$.

Then P_r is a suffix of T_{i+1}.

So P_{r-1} is a suffix of T_i.

So $\sigma(T_i) \geq r - 1 > q$.

Now, remind ourselves of what we are trying to do.

> Main Result: For each $i \leq n$, the value of q after the ith iteration of the main for loop in FINITE-AUTOMATON-MATCHING is $\sigma(T_i)$, i.e. the length of the longest prefix of the pattern P that is a suffix of T_i.

For the inductive step, suppose $q = \sigma(T_i)$ after the i'th iteration.

In the $(i+1)$st iteration, the new value of q is assigned to be $\delta(q, T[i+1])$.

We want to show that this is equal to $\sigma(T_{i+1})$.

From the lemma that we just proved, we can assume $\sigma(T_{i+1}) \leq q + 1$.

$\sigma(T_{i+1})$ is the length of the longest prefix of P that is a suffix of T_{i+1}.

Since $\sigma(T_{i+1}) \leq q+1$, we know that
$\sigma(T_{i+1}) = \sigma(T[i+1-q] \ldots T[i+1])$ where
$T[i+1-q] \ldots T[i+1]$ is the last $q+1$ characters of T_{i+1}.

But we know what the last $q+1$ characters of T_{i+1} are.
$T[i+1-q] \ldots T[i+1] = P_q T[i+1]$.

So $\sigma(T_{i+1}) = \sigma(P_q T[i+1]) = \delta(q, T[i+1])$, which is just what the algorithm does.

$\sigma(T_{i+1})$ is the length of the longest prefix of P that is a suffix of T_{i+1}.

Since $\sigma(T_{i+1}) \leq q+1$, we know that
$\sigma(T_{i+1}) = \sigma(T[i+1-q] \ldots T[i+1])$ where
$T[i+1-q] \ldots T[i+1]$ is the last $q+1$ characters of T_{i+1}.

But we know what the last $q+1$ characters of T_{i+1} are.
$T[i+1-q] \ldots T[i+1] = P_q T[i+1]$.

So $\sigma(T_{i+1}) = \sigma(P_q T[i+1]) = \delta(q, T[i+1])$, which is just what the algorithm does.

$\sigma(T_{i+1})$ is the length of the longest prefix of P that is a suffix of T_{i+1}.

Since $\sigma(T_{i+1}) \leq q+1$, we know that
$\sigma(T_{i+1}) = \sigma(T[i+1-q] \ldots T[i+1])$ where
$T[i+1-q] \ldots T[i+1]$ is the last $q+1$ characters of T_{i+1}.

But we know what the last $q+1$ characters of T_{i+1} are.
$T[i+1-q] \ldots T[i+1] = P_q T[i+1]$.

So $\sigma(T_{i+1}) = \sigma(P_q T[i+1]) = \delta(q, T[i+1])$, which is just what the algorithm does.

$\sigma(T_{i+1})$ is the length of the longest prefix of P that is a suffix of T_{i+1}.

Since $\sigma(T_{i+1}) \leq q+1$, we know that
$\sigma(T_{i+1}) = \sigma(T[i+1-q] \ldots T[i+1])$ where
$T[i+1-q] \ldots T[i+1]$ is the last $q+1$ characters of T_{i+1}.

But we know what the last $q+1$ characters of T_{i+1} are.
$T[i+1-q] \ldots T[i+1] = P_q T[i+1]$.

So $\sigma(T_{i+1}) = \sigma(P_q T[i+1]) = \delta(q, T[i+1])$, which is just what the algorithm does.

We have now proven the correctness of the following algorithm.

```
FINITE-AUTOMATON-MATCHING (T, δ, m)
    n ← length(T)
    q ← 0
    for i ← 1 to n
        q ← δ(q, T[i])
        if q = m
            print "pattern occurs with shift i − m"
```

But we have not shown how to compute the transition function δ.

Computing the transition function

The following procedure computes the transition function δ from a given pattern P and alphabet $\mathcal{A}$. Recall that $\delta(q, x)$ should be $\sigma(P_q x)$, the length of the longest pattern prefix that is a suffix of $P_q x$.

```
COMPUTE-TRANSITION-FUNCTION (P, A)
    m ← length(P)
    for q ← 0 to m
        for each x ∈ A
            k ← min(m + 1, q + 2)
            repeat k ← k − 1 until P_k is a suffix of P_q x
            δ(q, x) ← k
```

The running time is $O(m^3|\mathcal{A}|)$ ... why?

Complexity improvable to $\Theta(m|\mathcal{A}|)$, which is best possible.

Computing the transition function

The following procedure computes the transition function δ from a given pattern P and alphabet $\mathcal{A}$. Recall that $\delta(q, x)$ should be $\sigma(P_q x)$, the length of the longest pattern prefix that is a suffix of $P_q x$.

```
COMPUTE-TRANSITION-FUNCTION (P, A)
    m ← length(P)
    for q ← 0 to m
        for each x ∈ A
            k ← min(m + 1, q + 2)
            repeat k ← k − 1 until P_k is a suffix of P_q x
            δ(q, x) ← k
```

The running time is $O(m^3|\mathcal{A}|)$ ... why?

Complexity improvable to $\Theta(m|\mathcal{A}|)$, which is best possible.

Computing the transition function

The following procedure computes the transition function δ from a given pattern P and alphabet $\mathcal{A}$. Recall that $\delta(q, x)$ should be $\sigma(P_q x)$, the length of the longest pattern prefix that is a suffix of $P_q x$.

```
COMPUTE-TRANSITION-FUNCTION (P, A)
    m ← length(P)
    for q ← 0 to m
        for each x ∈ A
            k ← min(m + 1, q + 2)
            repeat k ← k − 1 until P_k is a suffix of P_q x
            δ(q, x) ← k
```

The running time is $O(m^3|\mathcal{A}|)$ ... why?

Complexity improvable to $\Theta(m|\mathcal{A}|)$, which is best possible.

Computing the transition function

The following procedure computes the transition function δ from a given pattern P and alphabet $\mathcal{A}$. Recall that $\delta(q, x)$ should be $\sigma(P_q x)$, the length of the longest pattern prefix that is a suffix of $P_q x$.

```
COMPUTE-TRANSITION-FUNCTION (P, A)
    m ← length(P)
    for q ← 0 to m
        for each x ∈ A
            k ← min(m + 1, q + 2)
            repeat k ← k − 1 until P_k is a suffix of P_q x
            δ(q, x) ← k
```

The running time is $O(m^3|\mathcal{A}|)$ ... why?

Complexity improvable to $\Theta(m|\mathcal{A}|)$, which is best possible.

The picture so far

- Defined the String Matching problem.
- Defined, implemented and seen examples of the brute-force algorithm. Time complexity $\Theta((n-m)m)$.
- Defined and seen examples of an alternative approach based on automata theory. Time complexity $O(n+m|\mathcal{A}|)$.
- Now we will look at the Knuth, Morris & Pratt algorithm. This algorithm is still based on the string-matching automaton approach but its time complexity is $O(n+m)$. The trick is to avoid the explicit computation of δ.

The picture so far

- Defined the String Matching problem.
- Defined, implemented and seen examples of the brute-force algorithm. Time complexity $\Theta((n-m)m)$.
- Defined and seen examples of an alternative approach based on automata theory. Time complexity $O(n+m|\mathcal{A}|)$.
- Now we will look at the Knuth, Morris & Pratt algorithm. This algorithm is still based on the string-matching automaton approach but its time complexity is $O(n+m)$. The trick is to avoid the explicit computation of δ.

The picture so far

- Defined the String Matching problem.
- Defined, implemented and seen examples of the brute-force algorithm. Time complexity $\Theta((n-m)m)$.
- Defined and seen examples of an alternative approach based on automata theory. Time complexity $O(n+m|\mathcal{A}|)$.
- Now we will look at the Knuth, Morris & Pratt algorithm. This algorithm is still based on the string-matching automaton approach but its time complexity is $O(n+m)$. The trick is to avoid the explicit computation of δ.

The picture so far

- Defined the String Matching problem.
- Defined, implemented and seen examples of the brute-force algorithm. Time complexity $\Theta((n-m)m)$.
- Defined and seen examples of an alternative approach based on automata theory. Time complexity $O(n+m|\mathcal{A}|)$.
- Now we will look at the Knuth, Morris & Pratt algorithm. This algorithm is still based on the string-matching automaton approach but its time complexity is $O(n+m)$. The trick is to avoid the explicit computation of δ.

Knuth, Morris, Pratt algorithm

The only inefficiency in the automaton algorithm is in the computation of the automaton itself.

The running time of Knuth, Morris, and Pratt's algorithm is linear in $n + m$. The algorithm uses an auxiliary function π, defined over the states of the automaton. The computation of π, given P, takes time $O(m)$.

Roughly speaking, $\pi(q)$ gives us enough information to quickly compute $\delta(q, x) = \sigma(P_q x)$ for any $x \in \mathcal{A}$.

Knuth, Morris, Pratt algorithm

The only inefficiency in the automaton algorithm is in the computation of the automaton itself.

The running time of Knuth, Morris, and Pratt's algorithm is linear in $n + m$. The algorithm uses an auxiliary function π, defined over the states of the automaton. The computation of π, given P, takes time $O(m)$.

Roughly speaking, $\pi(q)$ gives us enough information to quickly compute $\delta(q, x) = \sigma(P_q x)$ for any $x \in \mathcal{A}$.

Knuth, Morris, Pratt algorithm

The only inefficiency in the automaton algorithm is in the computation of the automaton itself.

The running time of Knuth, Morris, and Pratt's algorithm is linear in $n + m$. The algorithm uses an auxiliary function π, defined over the states of the automaton. The computation of π, given P, takes time $O(m)$.

Roughly speaking, $\pi(q)$ gives us enough information to quickly compute $\delta(q, x) = \sigma(P_q x)$ for any $x \in \mathcal{A}$.

Prefix function

The *prefix function* for a pattern P, is the function $\pi : \{1, \ldots, m\} \to \{0, \ldots, m-1\}$ such that

$$\pi[q] = \max\{k : P_k \text{ is a proper suffix of } P_q\}$$

Example. Let $P = 113\ 111\ 513\ 113$. The corresponding prefix function is

q	1	2	3	4	5	6	7	8	9	10	11	12
$P[q]$	1	1	3	1	1	1	5	1	3	1	1	3
$\pi[q]$	0	1	0	1	2	2	0	1	0	1	2	3

To define, say, $\pi[q]$ for $q = 6$ we consider $P_q \equiv 113\ 111$, and then all prefixes $P_{q-1}, P_{q-2}, \ldots, \varepsilon$. We find out that $P_2 (\equiv 11)$ is a suffix of P_q. Hence $\pi[6] = 2$.

Prefix function

The *prefix function* for a pattern P, is the function $\pi : \{1, \ldots, m\} \to \{0, \ldots, m-1\}$ such that

$$\pi[q] = \max\{k : P_k \text{ is a proper suffix of } P_q\}$$

Example. Let $P = 113\ 111\ 513\ 113$. The corresponding prefix function is

q	1	2	3	4	5	6	7	8	9	10	11	12
$P[q]$	1	1	3	1	1	1	5	1	3	1	1	3
$\pi[q]$	0	1	0	1	2	2	0	1	0	1	2	3

To define, say, $\pi[q]$ for $q = 6$ we consider $P_q \equiv 113\ 111$, and then all prefixes $P_{q-1}, P_{q-2}, \ldots, \varepsilon$. We find out that $P_2 (\equiv 11)$ is a suffix of P_q. Hence $\pi[6] = 2$.

Algorithm

```
KMP-MATCHING (T, P)
    n ← length(T)
    m ← length(P)
    π ← COMPUTE-PREFIX-FUNCTION (P)
    q ← 0
    for i ← 1 to n
        while (q > 0 ∧ P[q + 1] ≠ T[i])   q ← π[q]
        if (P[q + 1] = T[i])   q ← q + 1
        if q = m
            print "pattern occurs with shift i − m"
            q ← π[q]
```

Main Result: For each $i \leq n$, the value of q after the ith iteration of the main for loop is $\sigma(T_i)$, i.e. the length of the longest prefix of the pattern P that is a suffix of T_i.

```
COMPUTE-PREFIX-FUNCTION (P)
    m ← length(P)
    π[1] ← 0
    k ← 0
    for q ← 2 to m
        while (k > 0 ∧ P[k + 1] ≠ P[q])   k ← π[k]
        if (P[k + 1] = P[q])   k ← k + 1
        π[q] ← k
```

$\pi[q] = \max\{k : P_k \text{ is a proper suffix of } P_q\}$

Note: Before iteration q starts, the value of k is $\pi[q-1]$.

Example

Let $T =$ abdcababdcabdcb and $P =$ abdcabd.

We first compute the prefix function.

```
COMPUTE-PREFIX-FUNCTION (P)
    m ← length(P)
    π[1] ← 0
    k ← 0
    for q ← 2 to m
        while (k > 0 ∧ P[k + 1] ≠ P[q])   k ← π[k]
        if (P[k + 1] = P[q])   k ← k + 1
        π[q] ← k
```

q	1	2	3	4	5	6	7
$P[q]$	a	b	d	c	a	b	d
$\pi[q]$	0	0	0	0	1	2	3

Now given the prefix function

q	1	2	3	4	5	6	7
$P[q]$	a	b	d	c	a	b	d
$\pi[q]$	0	0	0	0	1	2	3

we simulate the algorithm

```
KMP-MATCHING (T, P)
    π ← COMPUTE-PREFIX-FUNCTION (P)
    q ← 0
    for i ← 1 to n
        while (q > 0 ∧ P[q + 1] ≠ T[i])   q ← π[q]
        if (P[q + 1] = T[i])   q ← q + 1
        if q = m
            print "pattern occurs with shift i − m"
            q ← π[q]
```

with text $T =$ abdcababdcabdcb, $n = 15$, $m = 7$.

The simulation starts like this before $i = 1$.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	a	b	d	c	a	b	d								
T	a	b	d	c	a	b	a	b	d	c	a	b	d	c	b
q	0														

Then $i = 1$ and $q = 0$ so the **while** loop is skipped, $P[q + 1]$ is equal to $T[i]$ so q becomes one, and we move to the $i = 2$ iteration

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	a	b	d	c	a	b	d								
T	a	b	d	c	a	b	a	b	d	c	a	b	d	c	b
q	0	1													
	1														

So starting from

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	a	b	d	c	a	b	d								
T	a	b	d	c	a	b	a	b	d	c	a	b	d	c	b
q	0	1													
	1														

$i = 2$, q IS positive, but $P[q + 1] = T[i]$ so the **while** loop is skipped again, and q is increased to two, and we move to the $i = 3$ iteration.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	a	b	d	c	a	b	d								
T	a	b	d	c	a	b	a	b	d	c	a	b	d	c	b
q	0	1	2												
	1	2													

$i = 3$, $i = 4$ up to $i = 6$ same story, q is successively increased, and each time we move to the next iteration.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	a	b	d	c	a	b	d								
T	a	b	d	c	a	b	a	b	d	c	a	b	d	c	b
q	0	1	2	3	4	5	6								
	1	2	3	4	5	6									

So starting from

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	a	b	d	c	a	b	d								
T	a	b	d	c	a	b	a	b	d	c	a	b	d	c	b
q	0	1	2	3	4	5	6								
	1	2	3	4	5	6									

$i = 7$, $q = 6$ is positive and $P[q+1] \neq T[i]$ (meaning we can't match P_7 with T_i) so we run $q \leftarrow \pi[q]$ inside the **while** loop to get $q = 2$. But we still have $P[q+1] \neq T[i]$ (meaning that we can't match P_3 with T_i so we run $q \leftarrow \pi[q]$ again to get $q = 0$. Now $P[q+1] = T[i]$ so q gets incremented.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	a	b	d	c	a	b	d								
T	a	b	d	c	a	b	a	b	d	c	a	b	d	c	b
q	0	1	2	3	4	5	6	1							
	1	2	3	4	5	6	2								
							0								
							1								

$i = 8$ up to $i = 12$, nothing exciting happens, q keeps increasing ...

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	a	b	d	c	a	b	d								
T	a	b	d	c	a	b	a	b	d	c	a	b	d	c	b
q	0	1	2	3	4	5	6	1	2	3	4	5	6		
	1	2	3	4	5	6	2	2	3	4	5	6			
							0								
							1								

... $i = 13$ again we skip the **while** loop and increase q and ... there is a match! So we run $q \leftarrow \pi[q]$ in the final **if** statement. Therefore q is set to three.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	a	b	d	c	a	b	d								
T	a	b	d	c	a	b	a	b	d	c	a	b	d	c	b
q	0	1	2	3	4	5	6	1	2	3	4	5	6	3	
	1	2	3	4	5	6	2	2	3	4	5	6	7		
							0						3		
							1								

$i = 14$, $P[q+1] = T[i]$, hence q is increased but for $i = 15$, q IS positive and $P[q+1] \neq T[i]$, hence we enter the **while** loop and reset q to zero ... and that's the end of it!

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	a	b	d	c	a	b	d								
T	a	b	d	c	a	b	a	b	d	c	a	b	d	c	b
q	0 1	1 2	2 3	3 4	4 5	5 6	6 2 0 1	1 2	2 3	3 4	4 5	5 6	6 7 3	3 4	4 0

Runinng Time Analysis - Amortized Analysis

COMPUTE-PREFIX-FUNCTION (P)

$m \leftarrow \text{length}(P)$

$\pi[1] \leftarrow 0$

$k \leftarrow 0$

for $q \leftarrow 2$ **to** m

while $(k > 0 \wedge P[k+1] \neq P[q])$ $k \leftarrow \pi[k]$

if $(P[k+1] = P[q])$ $k \leftarrow k+1$

$\pi[q] \leftarrow k$

Let k_i be value of k just before iteration $q = i$. Let c_i be cost of this iteration. Let $\hat{c}_i = c_i + k_{i+1} - k_i$.

Then $\sum_{i=2}^{m} \hat{c}_i = \sum_{i=2}^{m} c_i + k_{m+1} - k_2$.

$\pi(k)$ is the maximum j so that P_j is a proper suffix of P_k. So $0 \leq \pi(k) < k$ so $k_{m+1} \leq m$ and $k_2 = 0$. So it suffices to show that $\hat{c}_i = O(1)$.

Runinng Time Analysis - Amortized Analysis

COMPUTE-PREFIX-FUNCTION (P)

$m \leftarrow \text{length}(P)$

$\pi[1] \leftarrow 0$

$k \leftarrow 0$

for $q \leftarrow 2$ **to** m

while $(k > 0 \wedge P[k+1] \neq P[q])$ $k \leftarrow \pi[k]$

if $(P[k+1] = P[q])$ $k \leftarrow k+1$

$\pi[q] \leftarrow k$

Let k_i be value of k just before iteration $q = i$. Let c_i be cost of this iteration. Let $\hat{c}_i = c_i + k_{i+1} - k_i$.

Then $\sum_{i=2}^{m} \hat{c}_i = \sum_{i=2}^{m} c_i + k_{m+1} - k_2$.

$\pi(k)$ is the maximum j so that P_j is a proper suffix of P_k. So $0 \leq \pi(k) < k$ so $k_{m+1} \leq m$ and $k_2 = 0$. So it suffices to show that $\hat{c}_i = O(1)$.

Runinng Time Analysis - Amortized Analysis

COMPUTE-PREFIX-FUNCTION (P)

$m \leftarrow \text{length}(P)$

$\pi[1] \leftarrow 0$

$k \leftarrow 0$

for $q \leftarrow 2$ **to** m

while $(k > 0 \wedge P[k+1] \neq P[q])$ $k \leftarrow \pi[k]$

if $(P[k+1] = P[q])$ $k \leftarrow k+1$

$\pi[q] \leftarrow k$

Let k_i be value of k just before iteration $q = i$. Let c_i be cost of this iteration. Let $\hat{c}_i = c_i + k_{i+1} - k_i$.

Then $\sum_{i=2}^{m} \hat{c}_i = \sum_{i=2}^{m} c_i + k_{m+1} - k_2$.

$\pi(k)$ is the maximum j so that P_j is a proper suffix of P_k. So $0 \leq \pi(k) < k$ so $k_{m+1} \leq m$ and $k_2 = 0$. So it suffices to show that $\hat{c}_i = O(1)$.

Runinng Time Analysis - Amortized Analysis

COMPUTE-PREFIX-FUNCTION (P)

$m \leftarrow \text{length}(P)$

$\pi[1] \leftarrow 0$

$k \leftarrow 0$

for $q \leftarrow 2$ **to** m

while $(k > 0 \wedge P[k+1] \neq P[q])$ $k \leftarrow \pi[k]$

if $(P[k+1] = P[q])$ $k \leftarrow k+1$

$\pi[q] \leftarrow k$

Let k_i be value of k just before iteration $q = i$. Let c_i be cost of this iteration. Let $\hat{c}_i = c_i + k_{i+1} - k_i$.

Then $\sum_{i=2}^{m} \hat{c}_i = \sum_{i=2}^{m} c_i + k_{m+1} - k_2$.

$\pi(k)$ is the maximum j so that P_j is a proper suffix of P_k. So $0 \leq \pi(k) < k$ so $k_{m+1} \leq m$ and $k_2 = 0$. So it suffices to show that $\hat{c}_i = O(1)$.

Runinng Time Analysis - Amortized Analysis

COMPUTE-PREFIX-FUNCTION (P)

$m \leftarrow \text{length}(P)$

$\pi[1] \leftarrow 0$

$k \leftarrow 0$

for $q \leftarrow 2$ **to** m

while $(k > 0 \wedge P[k+1] \neq P[q])$ $k \leftarrow \pi[k]$

if $(P[k+1] = P[q])$ $k \leftarrow k+1$

$\pi[q] \leftarrow k$

Let k_i be value of k just before iteration $q = i$. Let c_i be cost of this iteration. Let $\hat{c}_i = c_i + k_{i+1} - k_i$.

Then $\sum_{i=2}^{m} \hat{c}_i = \sum_{i=2}^{m} c_i + k_{m+1} - k_2$.

$\pi(k)$ is the maximum j so that P_j is a proper suffix of P_k. So $0 \leq \pi(k) < k$ so $k_{m+1} \leq m$ and $k_2 = 0$. So it suffices to show that $\hat{c}_i = O(1)$.

Runinng Time Analysis - Amortized Analysis

COMPUTE-PREFIX-FUNCTION (P)

$m \leftarrow \text{length}(P)$

$\pi[1] \leftarrow 0$

$k \leftarrow 0$

for $q \leftarrow 2$ **to** m

while $(k > 0 \wedge P[k+1] \neq P[q])$ $k \leftarrow \pi[k]$

if $(P[k+1] = P[q])$ $k \leftarrow k+1$

$\pi[q] \leftarrow k$

Let k_i be value of k just before iteration $q = i$. Let c_i be cost of this iteration. Let $\hat{c}_i = c_i + k_{i+1} - k_i$.

Then $\sum_{i=2}^{m} \hat{c}_i = \sum_{i=2}^{m} c_i + k_{m+1} - k_2$.

$\pi(k)$ is the maximum j so that P_j is a proper suffix of P_k. So $0 \leq \pi(k) < k$ so $k_{m+1} \leq m$ and $k_2 = 0$. So it suffices to show that $\hat{c}_i = O(1)$.

Runinng Time Analysis - Amortized Analysis

COMPUTE-PREFIX-FUNCTION (P)

$m \leftarrow \text{length}(P)$

$\pi[1] \leftarrow 0$

$k \leftarrow 0$

for $q \leftarrow 2$ **to** m

while $(k > 0 \wedge P[k+1] \neq P[q])$ $k \leftarrow \pi[k]$

if $(P[k+1] = P[q])$ $k \leftarrow k+1$

$\pi[q] \leftarrow k$

Let k_i be value of k just before iteration $q = i$. Let c_i be cost of this iteration. Let $\hat{c}_i = c_i + k_{i+1} - k_i$.

Then $\sum_{i=2}^{m} \hat{c}_i = \sum_{i=2}^{m} c_i + k_{m+1} - k_2$.

$\pi(k)$ is the maximum j so that P_j is a proper suffix of P_k. So $0 \leq \pi(k) < k$ so $k_{m+1} \leq m$ and $k_2 = 0$. So it suffices to show that $\hat{c}_i = O(1)$.

Runinng Time Analysis - Amortized Analysis

COMPUTE-PREFIX-FUNCTION (P)

$m \leftarrow \text{length}(P)$

$\pi[1] \leftarrow 0$

$k \leftarrow 0$

for $q \leftarrow 2$ **to** m

while $(k > 0 \wedge P[k+1] \neq P[q])$ $k \leftarrow \pi[k]$

if $(P[k+1] = P[q])$ $k \leftarrow k+1$

$\pi[q] \leftarrow k$

Let k_i be value of k just before iteration $q = i$. Let c_i be cost of this iteration. Let $\hat{c}_i = c_i + k_{i+1} - k_i$.

Then $\sum_{i=2}^{m} \hat{c}_i = \sum_{i=2}^{m} c_i + k_{m+1} - k_2$.

$\pi(k)$ is the maximum j so that P_j is a proper suffix of P_k. So $0 \leq \pi(k) < k$ so $k_{m+1} \leq m$ and $k_2 = 0$. So it suffices to show that $\hat{c}_i = O(1)$.

Runinng Time Analysis - Amortized Analysis

COMPUTE-PREFIX-FUNCTION (P)

$m \leftarrow \text{length}(P)$

$\pi[1] \leftarrow 0$

$k \leftarrow 0$

for $q \leftarrow 2$ **to** m

while $(k > 0 \wedge P[k+1] \neq P[q])$ $k \leftarrow \pi[k]$

if $(P[k+1] = P[q])$ $k \leftarrow k+1$

$\pi[q] \leftarrow k$

Let k_i be value of k just before iteration $q = i$. Let c_i be cost of this iteration. Let $\hat{c}_i = c_i + k_{i+1} - k_i$.

Then $\sum_{i=2}^{m} \hat{c}_i = \sum_{i=2}^{m} c_i + k_{m+1} - k_2$.

$\pi(k)$ is the maximum j so that P_j is a proper suffix of P_k. So $0 \leq \pi(k) < k$ so $k_{m+1} \leq m$ and $k_2 = 0$. So it suffices to show that $\hat{c}_i = O(1)$.

Runinng Time Analysis - Amortized Analysis

COMPUTE-PREFIX-FUNCTION (P)

$m \leftarrow \text{length}(P)$

$\pi[1] \leftarrow 0$

$k \leftarrow 0$

for $q \leftarrow 2$ **to** m

while $(k > 0 \wedge P[k+1] \neq P[q])$ $k \leftarrow \pi[k]$

if $(P[k+1] = P[q])$ $k \leftarrow k+1$

$\pi[q] \leftarrow k$

Let k_i be value of k just before iteration $q = i$. Let c_i be cost of this iteration. Let $\hat{c}_i = c_i + k_{i+1} - k_i$.

Then $\sum_{i=2}^{m} \hat{c}_i = \sum_{i=2}^{m} c_i + k_{m+1} - k_2$.

$\pi(k)$ is the maximum j so that P_j is a proper suffix of P_k. So $0 \leq \pi(k) < k$ so $k_{m+1} \leq m$ and $k_2 = 0$. So it suffices to show that $\hat{c}_i = O(1)$.

Runinng Time Analysis - Amortized Analysis

COMPUTE-PREFIX-FUNCTION (P)

$m \leftarrow \text{length}(P)$

$\pi[1] \leftarrow 0$

$k \leftarrow 0$

for $q \leftarrow 2$ **to** m

while $(k > 0 \wedge P[k+1] \neq P[q])$ $k \leftarrow \pi[k]$

if $(P[k+1] = P[q])$ $k \leftarrow k+1$

$\pi[q] \leftarrow k$

Let k_i be value of k just before iteration $q = i$. Let c_i be cost of this iteration. Let $\hat{c}_i = c_i + k_{i+1} - k_i$. We need to show $\hat{c}_i = O(1)$.

Since $0 \leq \pi(k) < k$, every iteration of the while loop makes $k_{i+1} - k_i$ smaller by at least -1 so the contribution of the while loop to $\hat{c}_i$ is at most 0.

Runinng Time Analysis - Amortized Analysis

COMPUTE-PREFIX-FUNCTION (P)

$m \leftarrow \text{length}(P)$

$\pi[1] \leftarrow 0$

$k \leftarrow 0$

for $q \leftarrow 2$ **to** m

while $(k > 0 \wedge P[k+1] \neq P[q])$ $k \leftarrow \pi[k]$

if $(P[k+1] = P[q])$ $k \leftarrow k+1$

$\pi[q] \leftarrow k$

Let k_i be value of k just before iteration $q = i$. Let c_i be cost of this iteration. Let $\hat{c}_i = c_i + k_{i+1} - k_i$. We need to show $\hat{c}_i = O(1)$.

Since $0 \leq \pi(k) < k$, every iteration of the while loop makes $k_{i+1} - k_i$ smaller by at least -1 so the contribution of the while loop to $\hat{c}_i$ is at most 0.

Running Time Analysis

```
KMP-MATCHING (T, P)
    n ← length(T)
    m ← length(P)
    π ← COMPUTE-PREFIX-FUNCTION (P)
    q ← 0
    for i ← 1 to n
        while (q > 0 ∧ P[q + 1] ≠ T[i])   q ← π[q]
        if (P[q + 1] = T[i])   q ← q + 1
        if q = m
            print "pattern occurs with shift i − m"
            q ← π[q]
```

Similar amortized analysis. q_i is the value of q just before iteration i. c_i is the cost of iteration i. $\hat{c}_i = c_i + q_{i+1} - q_i$. It suffices to show $\hat{c}_i = O(1)$.

Exercises

1. Simulate the behaviour of the three algorithm we have considered on the pattern $P \equiv$ abc and the text $T =$ aabcbcbabcabcabc.
2. Count the number of instructions executed in each case and find out how the algorithms rank with respect to running time.
3. Repeat exercise 1 and 2 with the text $T =$ abababababababab. Comment on the results!