STRINGS: ALGORITHMS

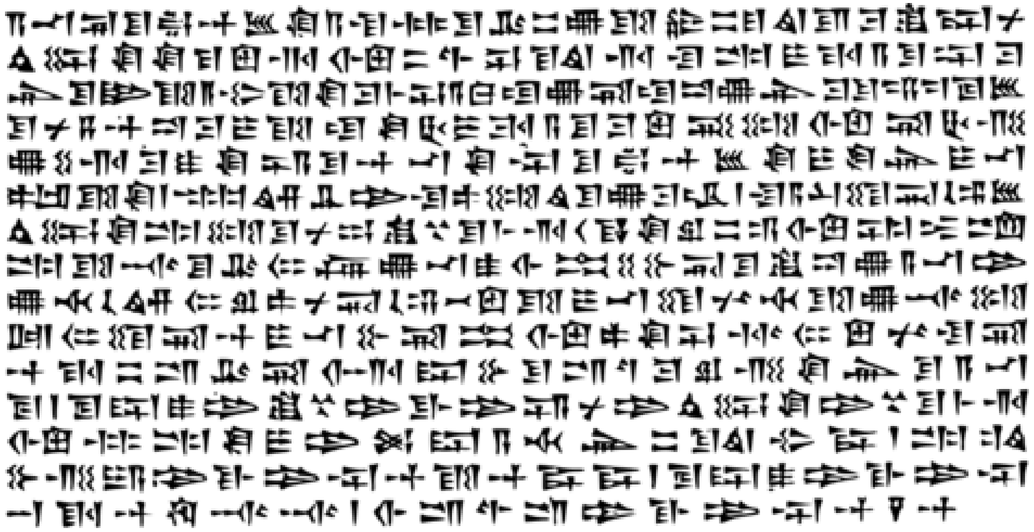
DATA STRUCTURES AND ALGORITHMS

STRINGS: ALGORITHMS

Content

ABCDEFGABCY

- Basic string snippets
- KMP algorithm
- Trie (prefix tree)
- Regular expressions



Αα Ββ Γγ Δδ Εε Ζζ
Ηη Θθ Ιι Κκ Λλ Μμ
Νν Ξξ Οο Ππ Ρρ Σσς
Ττ Υυ Φφ Χχ Ψψ Ωω

BASIC STRING ALGORITHMS

DATA STRUCTURES AND ALGORITHMS

BASIC STRING SNIPPETS

```c
#include <string.h>
#include <stdlib.h>

//remove specified characters from a string
void remchars(char *str, char c);

//remove specified chunks from a string
void remcnks(char *str, char *cnk);

//replace specified characters in a string
void replchars(char *str, char c1, char c2);

//replace specified chunks in a string (size-independent, just
remember about memory)
void replcnks(char *str, char *cnk1, char *cnk2);

//reverse a string
void reverse(char *str);
```

BASIC STRING SNIPPETS

```
//remove specified characters from a string
void remchars(char *str, char c)
{
    char *pos;
    while (pos = strchr(str, c))
        memmove(pos, pos + 1, strlen(pos));
}

//remove specified chunks from a string
void remcnks(char *str, char *cnk)
{
    char *pos;
    int clen = strlen(cnk);
    while (pos = strstr(str, cnk))
        memmove(pos, pos + clen, strlen(pos) - clen + 1);
}
```

BASIC STRING SNIPPETS

```
//replace specified characters in a string
void replchars(char *str, char c1, char c2)
{
    char *pos;
    while (pos = strchr(str, c1))
        *pos = c2;
}
//replace specified chunks in a string (size-independent, just remember about memory)
void replcnks(char *str, char *cnk1, char *cnk2)
{
    char *pos;
    int clen1 = strlen(cnk1), clen2 = strlen(cnk2);
    while (pos = strstr(str, cnk1))
    {
        memmove(pos + clen2, pos + clen1, strlen(pos) - clen1 + 1);
        memcpy(pos, cnk2, clen2);
    }
}
```

KPM ALGORITHM

DATA STRUCTURES AND ALGORITHMS

PREFIX FUNCTION KNUTH-MORRIS-PRATT ALGORITHM

Prefix function definition

- You are given a string s of length n s[0..n−1]. Prefix function for this string is defined as an array π of length n π[0..n−1], where π[i] is the length of the longest proper prefix of a substring s[0...i] which is also a suffix of this substring. A proper prefix of a string is a prefix that is not equal to the string itself. By definition, π[0]=0.

$$\pi[i] = \max_{k=0\ldots i}\{\, k\colon s[0\ldots k-1] = s[i-k+1\ldots i]\}$$

Example

"abcabcd": [0, 1, 0, 1, 2, 2, 3]

- "a" - no prefix == suffix
- "ab" - no prefix == suffix
- "abc" – no prefix==suffix
- "abca" - 1
- "abcab" - 2
- "abcabc" - 3
- "abcabcd" - 0

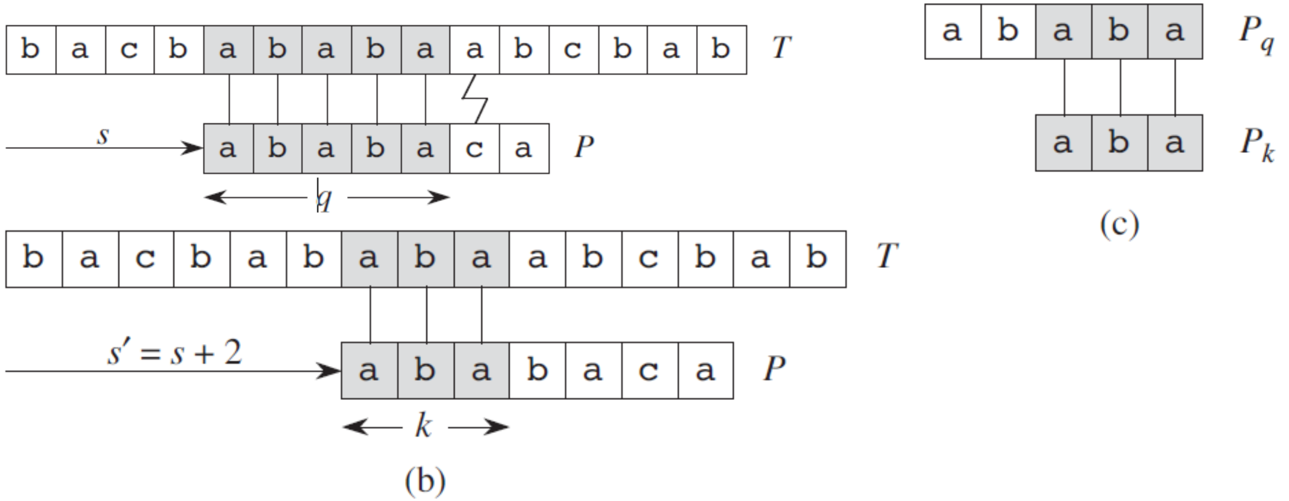
TRIVIAL ALGORITHM

$O(n^3)$

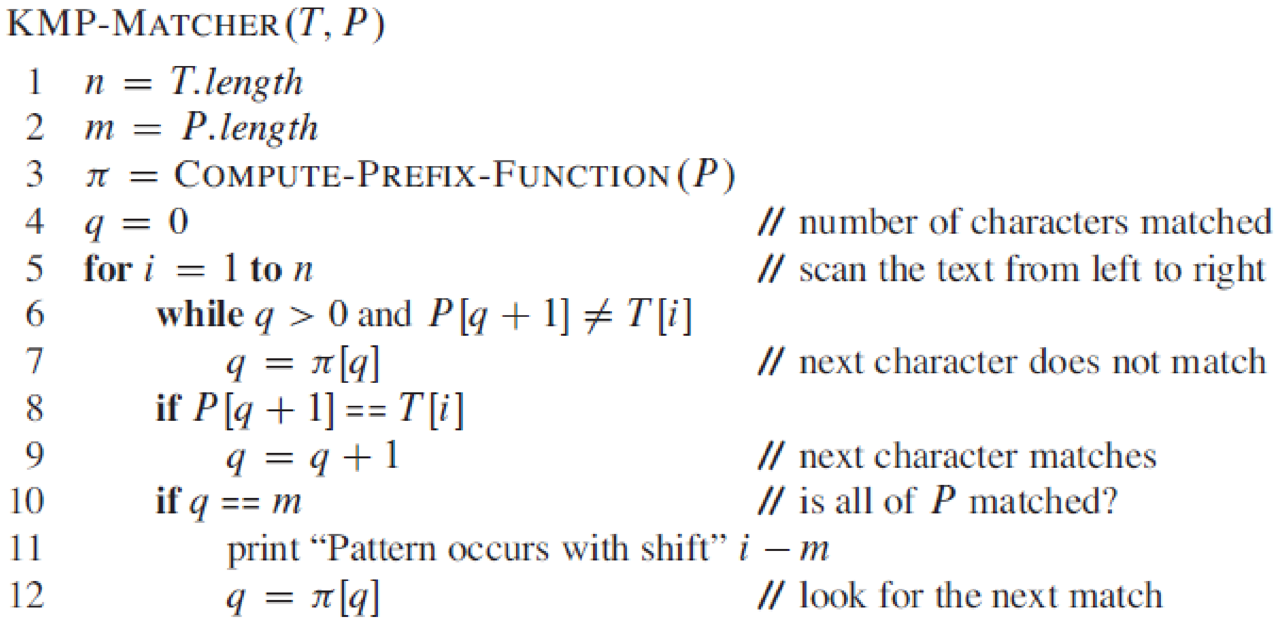
```
vector<int> prefix_function (string s) {

  int n = (int) s.length();
  vector<int> pi (n);
  for (int i=0; i<n; ++i)
    for (int k=0; k<=i; ++k)
      if (s.substr(0,k) == s.substr(i-k+1,k))
      pi[i] = k;
  return pi;
}
```

KMP IDEA

- The prefix function π. (a) The pattern P D ababaca aligns with a text T so that the first q = 5 characters match. Matching characters, shown shaded, are connected by vertical lines. (b) Using only our knowledge of the 5 matched characters, we can deduce that a shift of s + 1 is invalid, but that a shift of s’ = s+2 is consistent with everything we know about the text and therefore is potentially valid. (c)We can precompute useful information for such deductions by comparing the pattern with itself. Here, we see that the longest prefix of P that is also a proper suffix of P5 is P3. We represent this precomputed information in the array π, so that $\pi[5] = 3$. Given that q characters have matched successfully at shift s, the next potentially valid shift is at $s' = s + (q - \pi[q])$ as shown in part (b)



(b)



```
KMP-MATCHER(T, P)
n = T.length
m = P.length
π = COMPUTE-PREFIX-FUNCTION(P)
q = 0                                   // number of characters matched
for i = 1 to n                          // scan the text from left to right
    while q > 0 and P[q + 1] ≠ T[i]
        q = π[q]                        // next character does not match
    if P[q + 1] == T[i]
        q = q + 1                       // next character matches
    if q == m                           // is all of P matched?
        print "Pattern occurs with shift" i − m
        q = π[q]                        // look for the next match
```

KMP ALGORITHM

$$O(n)$$

```
vector<int> prefix_function (string s) {

    int n = (int) s.length();
    vector<int> pi (n);
    for (int i=1; i<n; ++i) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j]) ++j;
        pi[i] = j;
    }
    return pi;
}
```

This is an online algorithm, i.e. it processes the data as it arrives - for example, you can read the string characters one by one and process them immediately, finding the value of prefix function for each next character. The algorithm still requires storing the string itself and the previously calculated values of prefix function, but if we know beforehand the maximum value M the prefix function can take on the string, we can store only M+1 first characters of the string and the same number of values of the prefix function.

PREFIX TREE (TRIE)

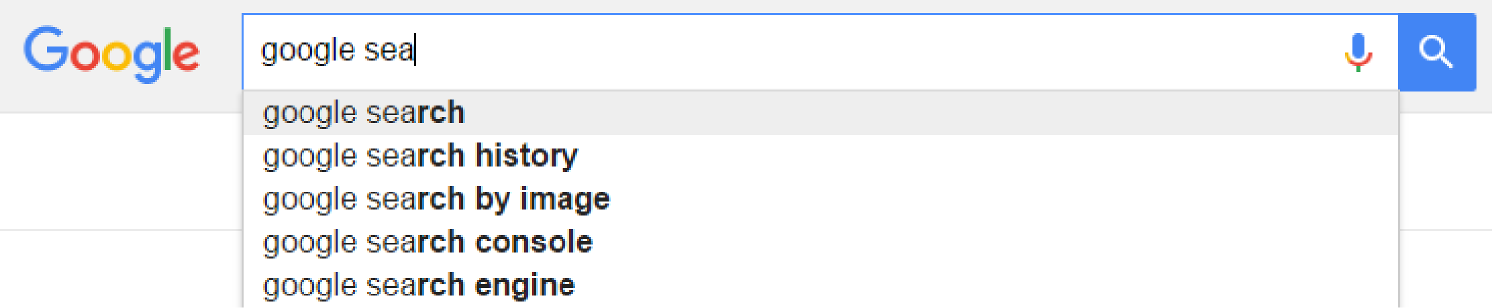
DATA STRUCTURES AND ALGORITHMS

DATA STRUCTURE

Prefix tree (a.k.a. Trie)

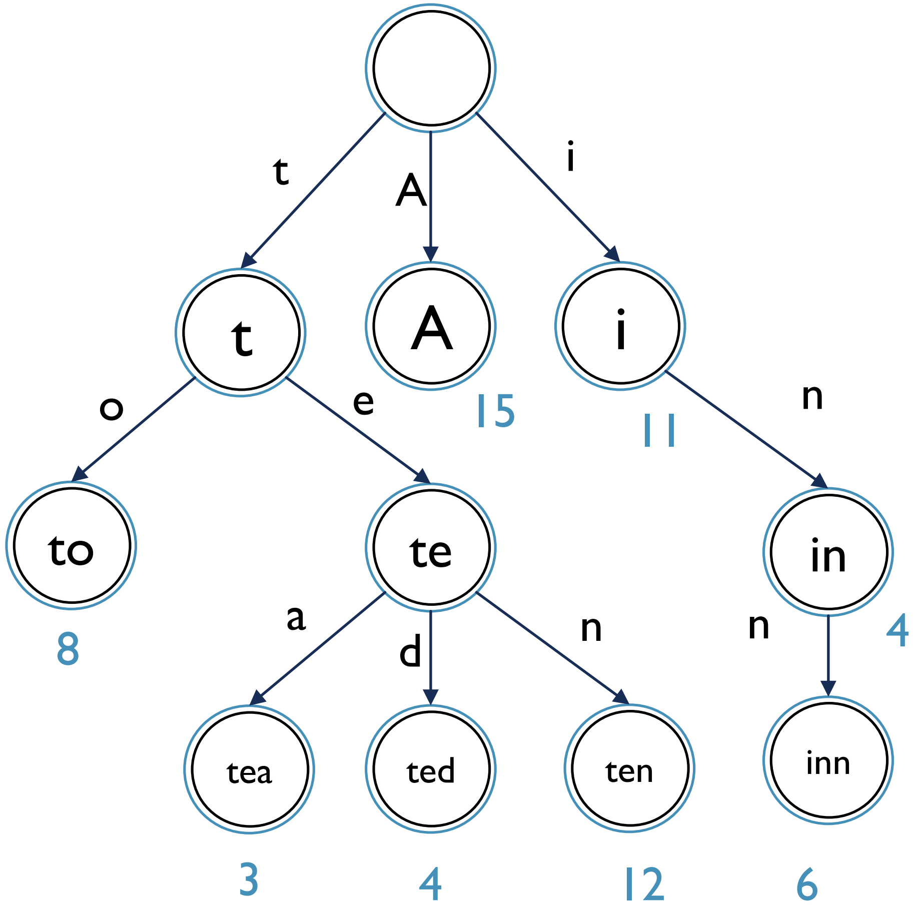
Trie data structure, or prefix tree (as they can be searched by prefixes), is a kind of search tree—an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated

Words	Count
A	15
I	11
In	4
inn	6
ten	12
ted	4
tea	3
to	8



google sea

google search
google search history
google search by image
google search console
google search engine



IMPLEMENTATION

```
struct node {

    node *next[26];           // pointers’ array
                              //next[i] - the next item ('a' + i)
    int strings;              //number of strings

    node() {
        for (int i = 0; i < 26; i++) {
            //initializing next[i] with nulls
            next[i] = nullptr;
        }
        strings = 0;
    }
};
```

IMPLEMENTATION

```
node *root = new node(); // the root represented with empty node.

void add(const string& s) {
    node *cur_v = root;     //current root

    for (int i = 0; i < s.length(); i++) {
        char c = s[i];
        if (cur_v->next[c - 'a'] == nullptr) {
            cur_v->next[c - 'a'] = new node();
        }
    cur_v = cur_v->next[c - 'a’];
    }

    cur_v->strings++;
}
```

IMPLEMENTATION

```
bool has(const string& s) {

    node *cur_v = root;

    for (int i = 0; i < s.length(); i++) {
        cur_v = cur_v->next[s[i] - 'a’];
        if (cur_v == nullptr) {
            return false;
        }
    }

    return cur_v->strings > 0;
}
```

IMPLEMENTATION

```
string cur_str = "";

void write(node *v = root) {
    for (int i = 0; i < v->strings; i++) {
        cout << cur_str << endl;
    }

    for (int i = 0; i < 26; i++) {
        if (v->next[i] != nullptr) {
            cur_str.push_back('a' + i);
            write(v->next[i]);
            cur_str.pop_back();
        }
    }
}
```

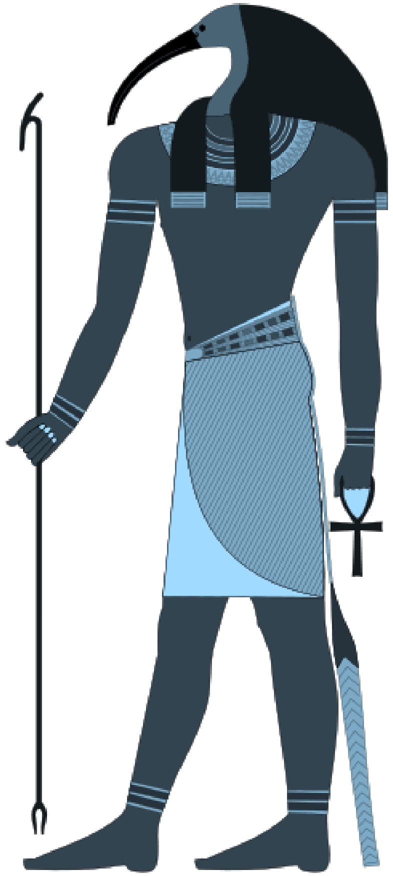
This method uses DFS
to print all the string from
Prefix tree

REGULAR EXPRESSIONS

DATA STRUCTURES AND ALGORITHMS

REGULAR EXPRESSIONS

#include<regex>



- Purpose: Build the expression by using string patterns
- Produce the **method** on **string** for processing

regex_replace

regex_match

regex_search

- Regular expression concept used in software developing
- Regex Grammars
  - ECMAScript (by default in C++)
  - Basic
  - Extended
  - Awk
  - grep
  - egrep

REGULAR EXPRESSIONS

```
#include<iostream>
#include<regex>
using namespace std;

int main() {
    string str;
    while (true)
    {
        cin >> str;
        regex myexpression("hello");

        bool is_match = regex_match(str, myexpression);

        if (is_match)
            cout << "matched";
        else
            cout << "not matched";
    }
}
```

REGULAR EXPRESSION

Now check is true, if it doesn't matter upper or lower case

```
regex myexpression("hello", regex_constants::icase);

bool is_match = regex_match(str, myexpression);
```

hello	matched
HELLO	matched
HeLlo	matched
hellfnjdg	not matched
hel	not matched

hello. – means any word start with hello + any symbol (except \n)

```
regex myexpression("hello.", regex_constants::icase);

bool is_match = regex_match(str, myexpression);
```

hello	matched
Helloh	matched
HELLOX	matched
hellfnjdg	not matched
hel	not matched

REGULAR EXPRESSIONS

0 or 1 - character (that has been precedes)

```
regex myexpression("hello?", regex_constants::icase);
bool is_match = regex_match(str, myexpression);
```

hellox	not matched
heLLo	matched
Hell	matched
hel	not matched

```
regex myexpression("hello*");
bool is_match = regex_match(str, myexpression);
```

hellooooo	matched
hello	matched
helloe	not matched
hell	not matched

REGULAR EXPRESSIONS

1 or more prev. characters

```
regex myexpression("hello+");
bool is_match = regex_match(str, myexpression);
```

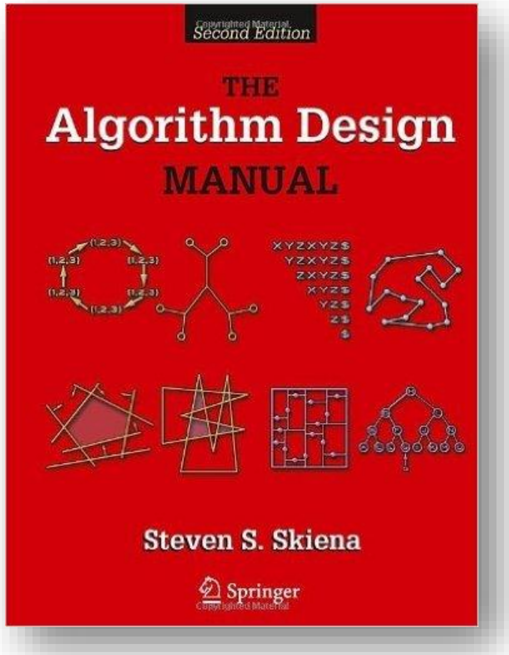
hellоооо matched
hello matched
hell not matched
hel not matched

[ ] – takes several characters as one characters
for this case l or o that are in brackets and *

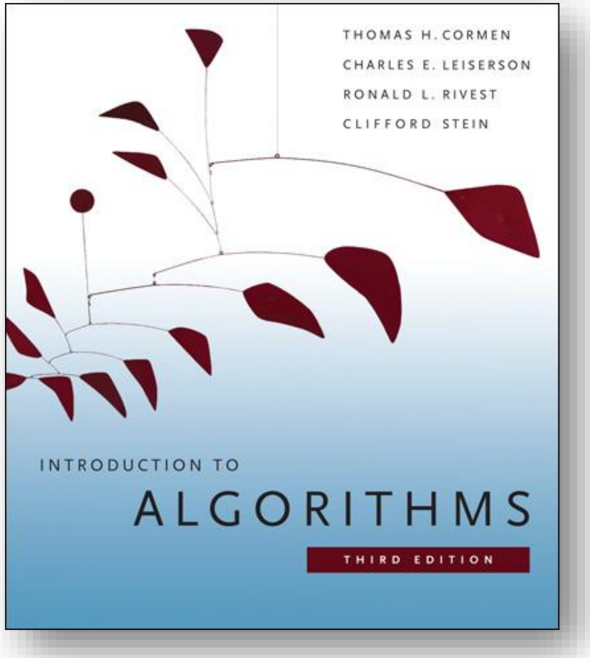
```
regex myexpression("hel[lo]*");
bool is_match = regex_match(str, myexpression);
```

hellooooo matched
hellollllo matched
hellollololll matched
hellli not matched

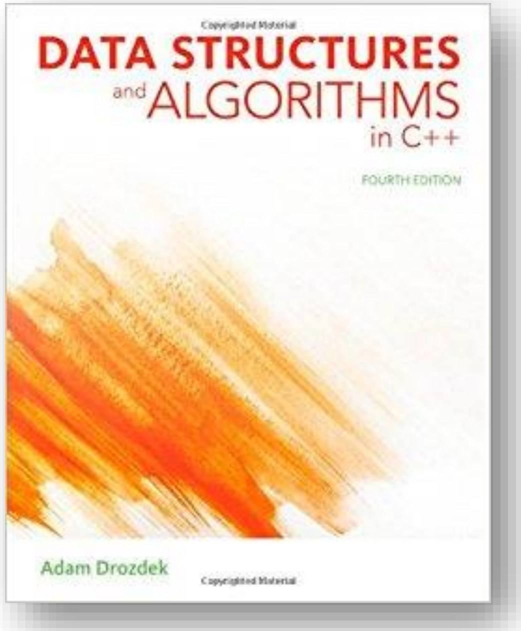
LITERATURE



Stieven Skienna
Algorithms design manual
Chapter 18: Set and String Problems (String Matching)
Page 628



Thomas H. Cormen
Introduction to Algorithms
Chapter VII, 32 String Matching (KPM algorithm)
Page 1002.



Adam Drozdek
Data structures and Algorithms in C++
Chapter 13: String Matching
Page 674