Outline

- The Basic Concept of Strings
- The Storage Structure of Strings
- The Implementation of Strings’ Operations
- Pattern Matching for Strings
  - Naïve Algorithm
  - KMP Fast String Matching
4.1 The Basic Concept of Strings

Examples for Strings

- s1=“123”
- s2=“ABBABBC”
- s3=“BB”
- s4=“BB”
- s5=“Hello World!”
- s5=“”
4.1 The Basic Concept of Strings

• Strings, a special kind of Linear Lists, whose element is Character
• Finite Sequence with n (≥ 0) characters, n ≥ 1, noted as S: “c_0c_1c_2…c_{n-1}”
  - S is the name of String
  - “c_0c_1c_2…c_{n-1}” is the value of String
  - c_i is the character in String
  - N is the length of the string: The number of characters in a String.

• Empty String: its length is 0, and don’t contain any characters. (Note its difference with ”Blank String”)

-
4.1 The Basic Concept of Strings

String - a special kind of Linear Lists

- Data Object
  - No special limitation
  - Data Object of a String is called as Character Set

- Basic Operation
  - The operating object of most Linear Lists is "element unit"
  - Strings always use "the whole string" as the operating object

- The storage method of Linear Lists can be applied in Strings
  - Choose storage method according to different situations
4.1 The Basic Concept of Strings

Char

• Char: the basic unit of Strings
• The value of the char depends on char set $\Sigma$ (same as linear Lists, finite set of nodes)
  - Binary char set: $\Sigma = \{0, 1\}$
  - DNA char set: $\Sigma = \{A, C, G, T\}$
  - English char set: $\Sigma = \{26$ lowercase letters, 26 uppercase letters, punctuation$\}$
  - ......
4.1 The Basic Concept of Strings

Char Encoding

• A byte (8 bits)
  - Encode 128 symbols using ASCII code System
  - Used both in C/C++

• Other Encodings
  - GB
  - CJK
  - UNICODE
The order for Char Encoding

- For facilitating comparison and operations of Strings, the Character code table follow “Partial order encoding rules”

- **Char Partial Order**: based on the natural meaning of char, some characters can be compared.
  - In most case characters are compared based on lexicographical
  - There are some exceptions in Chinese string, such as the “stroke” order
Data Type of strings

- It changed with the language
  - Simple Type
  - Composite Type

- Literals and Variables of String
  - String literals
    - Such as: “\n”, “a”, “student”...
  - String Variables
### 4.1 The Basic Concept of Strings

**Substring**

- **The definition of substring**
  
  Suppose there are two strings: $s_1, s_2$
  
  $s_1 = a_0 a_1 a_2 ... a_{n-1}$
  
  $s_2 = b_0 b_1 b_2 ... b_{m-1}$ where $0 \leq m \leq n$
  
  If there exist an integer $i$ ($0 \leq i \leq n-m$), such that $b_j = a_{i+j}$ for $j = 0, 1, ..., m-1$, then we call that $s_2$ is a **substring** of $s_1$, and $s_1$ is the **master string** of $s_2$, or we can call that $s_1$ contains $s_2$

- **Special substring**
  
  - **Empty string** is a substring of all strings
  
  - Any string $S$ is a substring of $S$ itself.
  
  - Proper substring: neither an empty string nor the master string
Chapter Four

Strings

4.1 The Basic Concept of Strings

The basic operation of String

C standard library needs \#include <string.h>

- Get the length of string \( \text{int strlen(char *s);} \)
- Copy operation of string \( \text{char *strcpy(char *s1, char*s2);} \)
- Concatenation operation of string \( \text{char *strcat(char *s1, char *s2);} \)
- Comparison of string (be careful)
  - \( \text{int strcmp(char *s1, char *s2);} \)
  - It depends on ASCII code, if \( s1 > s2 \), return value that is greater than 0; if two strings are equal, return 0
- Positioning \( \text{char *strchr(char *s, char c);} \)
- Right Positioning \( \text{char * strrchr(char *s, char c);} \)
- Get substring \( \text{char *strstr(const char *str1, const char *str2);} \)
4.1 The Basic Concept of Strings

Example for Positioning function

- String s:

\[
\begin{array}{ccccccccccc}
H & e & l & l & o & w & o & r & l & d & \0
\end{array}
\]

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
</table>

- Search char ‘o’, strchr(s,’o’) return 4
- Search char ‘r’ in the opposite direction, strrchr(s,’o’) return 7
Chapter Four

Strings

4.1 The Basic Concept of Strings

String Abstract Data Type

C++ standard string library

#include <string>
using namespace std;

• class String
  - Adapt to dynamic changes of the length of string.
  - use a dynamic variable-length storage structure.
## List of C++ String Some Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Substr</td>
<td>substr()</td>
<td>Return substring</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Copy/Swap</td>
<td>swap()</td>
<td>Exchange contents of two strings</td>
</tr>
<tr>
<td></td>
<td>copy()</td>
<td>Copy a string to another one</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assign</td>
<td>assign()</td>
<td>Assign a string, a char, a substring to another one.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Assign a string, a char to another one</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>insert/append</td>
<td>insert()</td>
<td>Insert a char, many chars or a string in the given position.</td>
</tr>
<tr>
<td></td>
<td>append() / +=</td>
<td>Append one or more chars, or string to the tail of another string</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concatenation</td>
<td>+</td>
<td>By placing a string behind the other to build a new one</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Find</td>
<td>find()</td>
<td>Find and return the start position of a substring</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replace/Clear</td>
<td>replace()</td>
<td>Replace a specified char or substring</td>
</tr>
<tr>
<td></td>
<td>clear()</td>
<td>Clear all char in the string</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statics</td>
<td>size() / length()</td>
<td>Return the number of chars in the string</td>
</tr>
<tr>
<td></td>
<td>max_size()</td>
<td>Return the max size of the string</td>
</tr>
</tbody>
</table>
4.1 The Basic Concept of Strings

Get char in String

- Overloaded subscript operator[]
  ```cpp
  char& string::operator [] (int n);
  ```
- Positioning subscript by Char
  ```cpp
  int string::find(char c, int start=0);
  ```
- Search in the opposite direction, locate the subscript of given char.
  ```cpp
  int string::rfind(char c, int pos=0);
  ```


1. Judge which is a substring of “software”
   - Empty string, “software”, “soft”, “oft”...
   - “fare”, “sfw”...

2. If string s=“software”, what’s the total number of all its substrings?
Outline

• The Basic Concept of Strings
• The Storage Structure of Strings
  - Sequential storage for string
  - The storage structure of class String
• The Implementation of Strings’ Operations
  - The implementation of Strings’ Operations
  - String class’ implementation
• Pattern Matching for Strings
  - Naïve Algorithm
  - KMP Fast String Matching
4.2 The storage structure of string and its implementation

Sequential storage for String

- For strings that has little change on the length, there are three processing scheme:
  1. Use S[0] to record the strength of string (Pascal)
     - Disadvantage: limit the maximum length of string and make sure that it doesn’t exceed 256.
  2. To store the length of string, open up a new place
     - Disadvantage: The maximum length of string is static, not space dynamically applied
  3. End with a special mark ‘\0’ (C/C++)
     - For example: C/C++ string standard library ( #include <string.h> )
     - ‘\0’ ASCII char table numbered 0, equivalent to the constant NULL, digit 0, constant false
The storage structure for String class

```cpp
private:  // implementation for string' storage structure
char *str; // data representation for string
int size;  // current length of string
```

For example,
String s1 = "Hello";

```
private:
    char *str;
    int size; // value is 5

......
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hello</td>
<td>\0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
The implementation of string operation

• String Length
  - int strlen(char *s);

• String Copy
  - char *strcpy(char *s1, char*s2);

• String concatenation
  - char *strcat(char *s1, char *s2);

• String Comparison
  - int strcmp(char *s1, char *s2);
4.2 The storage structure of string and its implementation

The implementation of string operation

// Get the length of string
int strlen(char d[]) {
    int i = 0;
    while (d[i] != '\0')
        i++;
    return i;
}
The implementation of string operation

// String copy
char *strcpy(char *d, char *s) {
    int i = 0;
    while (s[i] != '\0') {
        d[i] = s[i]; i++;
    }
    d[i] = '\0';
    return d;
}
The implementation of string operation

// String comparison
int strcmp(const char *s1, const char *s2) {
    int i = 0;
    while (s2[i] != '\0' && s1[i] != '\0') {
        if (s1[i] > s2[i])
            return 1;
        else if (s1[i] < s2[i])
            return -1;
        i++;
    }
    if (s1[i] == '\0' && s2[i] != '\0')
        return -1;
    else if (s2[i] == '\0' && s1[i] != '\0')
        return 1;
    return 0;
}
More simple algorithm

```c
int strcmp_1(char *s1, char *s2) {
    int i;
    for (i = 0; s1[i] == s2[i]; ++i) {
        if(s1[i] == '\0' && s2[i] == '\0')
            return 0; // if s1==s2
    }
    // if s1!=s2, then compare the first different char
    return (s1[i]-s2[i]) / abs(s1[i]-s2[i]);
}
```
The implementation of string operation

// constructor
String::String(char *s) {
    // Firstly, confirm the storage space needed for the new-built string,
    // the type of s is (char *)
    // As the initialized value of the new-built string.
    // Get the length of S by the standard function of string class.
    // strlen(s) get length of string
    size = strlen(s);
    // then, open up a space in the dynamic storage area
    // to store the initialized value of s, the end chars are also included.
    str = new char [size + 1];
    // When opening up space fails, it is abnormal, then quit
    assert(str != NULL);
    // use standard function strcpy to copy the whole string
    // to the storage space where the pointer of str points to
    strcpy(str, s);
}
The implementation of string operation

// Destructor
String::~String() {
    // Dynamic storage space must be released
    delete [] str;
}
The implementation of string operation

// Assign function
String String::operator= (String& s) {
    // String s will be assigned to this string
    // if the length of this string is not equal to the length of s, release this string
    // open up new spaces
    if (size != s.size) {
        delete [] str; // release storage space
        str = new char [s.size+1];
        // if opening up the dynamic storage space fails, then quit.
        assert(str != NULL);
        size = s.size;
    }
    strcpy(str, s.str);
    // return this instance as the instance of string class
    return *this;
}
4.2 The storage structure of string and its implementation

Substring extraction

\[
\text{\texttt{s2 = s1.Substr(6, 5);}}
\]
Exercises

• Given strings S1 and S2, please list all possible conditions that $S1+S2==S2+S1$ (‘+’ is the concatenation operation)

• Design an algorithm to reverse the characters in a given string, without extra storage for an intermediate string.
Outline

- The Basic Concept of Strings
- The Storage Structure of Strings
- The Implementation of Strings’ Operations
- Pattern Matching for Strings
  - Naïve algorithm
  - KMP Fast String Matching
4.3 Pattern Matching for Strings

• pattern matching
  - A target object $T$ (String)
  - (pattern) $P$ (String)
  The process of searching pattern $P$ that is given in target string.

• Application
  - Lookup of specified word, sentence in text editing.
    - UNIX/Linux: sed, awk, grep
  - Extraction of DNA information
  - Confirm the existence of a structure...

• Pattern set
4.3 Pattern Matching for Strings

Pattern Matching for Strings

- With a given pattern $P$, search a substring that is matched completely with pattern $p$ in target string $T$, and find the first substring that is matched completely with pattern $p$ (referred to as “Matching Strings”), return the position of its first character.

$$ T \quad t_0 \quad t_1 \quad \ldots \quad t_i \quad t_{i+1} \quad t_{i+2} \quad \ldots \quad t_{i+m-2} \quad t_{i+m-1} \quad \ldots \quad t_{n-1} $$

$$ \| \quad \| \quad \| \quad \| \quad \| $$

$$ P \quad p_0 \quad p_1 \quad p_2 \quad \ldots \quad p_{m-2} \quad p_{m-1} $$

To make pattern $P$ and target $T$ match, must meet:

$$ p_0 \quad p_1 \quad p_2 \ldots \quad p_{m-1} = t_i \quad t_{i+1} \quad t_{i+2} \ldots \quad t_{i+m-1} $$
4.3 Pattern Matching for Strings

Pattern Matching for Strings

- Goal: in ‘big’ text (such as, sentence, paragraph, or a book), to find/position a specific pattern
- Algorithm
  - Naive ("Brute Force")
  - Knuth-Morrit-Pratt (KMP)
  - ......
4.3 Pattern Matching for Strings

Pattern Matching for Strings (Brute Force)

- Set $T = t_0 t_1, t_2, ..., t_{n-1}$, $P = p_0, p_1, ..., p_{m-1}$
  - $i$ is subscript of $T$’s characters, $j$ is subscript of $P$’s characters.
  - **Successful match** ($p_0 = t_i$, $p_1 = t_{i+1}$, ..., $p_{m-1} = t_{i+m-1}$)
    - That’s, $T.substr(i, m) == P.substr(0, m)$
  - **Failed match** ($p_j \neq t_i$)
    - move $P$ to the right, and do comparison
- Try all possible cases
Example 1 for naïve matching

$T = \text{a a a a a a a a a a a a a a a a b}$

$P = \text{a a a a a a a a a a a a a a a a b}$
Example 2 for naïve matching

<table>
<thead>
<tr>
<th>T=</th>
<th>a b a b a b a b a b a b b</th>
</tr>
</thead>
<tbody>
<tr>
<td>P=</td>
<td>a b a b a b X</td>
</tr>
</tbody>
</table>

Matching process:

1. Compare T[0] with P[0]: Match
2. Compare T[1] with P[0]: Mismatch
3. Compare T[2] with P[0]: Match
5. Compare T[4] with P[0]: Match
7. Compare T[6] with P[0]: Match
8. Compare T[7] with P[0]: Mismatch
9. Compare T[8] with P[0]: Match
10. Compare T[9] with P[0]: Mismatch
11. Compare T[10] with P[0]: Match
13. Compare T[12] with P[0]: Match
15. Compare T[14] with P[0]: Match
17. Compare T[16] with P[0]: Match
18. Compare T[17] with P[0]: Mismatch
19. Compare T[18] with P[0]: Match
20. Compare T[19] with P[0]: Mismatch
21. Compare T[20] with P[0]: Match
22. Compare T[21] with P[0]: Mismatch
23. Compare T[22] with P[0]: Match
24. Compare T[23] with P[0]: Mismatch
25. Compare T[24] with P[0]: Match
27. Compare T[26] with P[0]: Match
28. Compare T[27] with P[0]: Mismatch
29. Compare T[28] with P[0]: Match
30. Compare T[29] with P[0]: Mismatch
31. Compare T[30] with P[0]: Match
32. Compare T[31] with P[0]: Mismatch
33. Compare T[32] with P[0]: Match
34. Compare T[33] with P[0]: Mismatch
35. Compare T[34] with P[0]: Match
36. Compare T[35] with P[0]: Mismatch
37. Compare T[36] with P[0]: Match
38. Compare T[37] with P[0]: Mismatch
39. Compare T[38] with P[0]: Match
40. Compare T[39] with P[0]: Mismatch
41. Compare T[40] with P[0]: Match
42. Compare T[41] with P[0]: Mismatch
43. Compare T[42] with P[0]: Match
44. Compare T[43] with P[0]: Mismatch
45. Compare T[44] with P[0]: Match
46. Compare T[45] with P[0]: Mismatch
47. Compare T[46] with P[0]: Match
48. Compare T[47] with P[0]: Mismatch
49. Compare T[48] with P[0]: Match
50. Compare T[49] with P[0]: Mismatch
51. Compare T[50] with P[0]: Match
52. Compare T[51] with P[0]: Mismatch
53. Compare T[52] with P[0]: Match
54. Compare T[53] with P[0]: Mismatch
55. Compare T[54] with P[0]: Match
56. Compare T[55] with P[0]: Mismatch
57. Compare T[56] with P[0]: Match
58. Compare T[57] with P[0]: Mismatch
59. Compare T[58] with P[0]: Match
60. Compare T[59] with P[0]: Mismatch
61. Compare T[60] with P[0]: Match
62. Compare T[61] with P[0]: Mismatch
63. Compare T[62] with P[0]: Match
64. Compare T[63] with P[0]: Mismatch
65. Compare T[64] with P[0]: Match
66. Compare T[65] with P[0]: Mismatch
67. Compare T[66] with P[0]: Match
68. Compare T[67] with P[0]: Mismatch
69. Compare T[68] with P[0]: Match
70. Compare T[69] with P[0]: Mismatch
71. Compare T[70] with P[0]: Match
72. Compare T[71] with P[0]: Mismatch
73. Compare T[72] with P[0]: Match
74. Compare T[73] with P[0]: Mismatch
75. Compare T[74] with P[0]: Match
76. Compare T[75] with P[0]: Mismatch
77. Compare T[76] with P[0]: Match
78. Compare T[77] with P[0]: Mismatch
79. Compare T[78] with P[0]: Match
80. Compare T[79] with P[0]: Mismatch
81. Compare T[80] with P[0]: Match
82. Compare T[81] with P[0]: Mismatch
83. Compare T[82] with P[0]: Match
84. Compare T[83] with P[0]: Mismatch
85. Compare T[84] with P[0]: Match
86. Compare T[85] with P[0]: Mismatch
87. Compare T[86] with P[0]: Match
88. Compare T[87] with P[0]: Mismatch
89. Compare T[88] with P[0]: Match
90. Compare T[89] with P[0]: Mismatch
91. Compare T[90] with P[0]: Match
92. Compare T[91] with P[0]: Mismatch
93. Compare T[92] with P[0]: Match
94. Compare T[93] with P[0]: Mismatch
95. Compare T[94] with P[0]: Match
96. Compare T[95] with P[0]: Mismatch
97. Compare T[96] with P[0]: Match
98. Compare T[97] with P[0]: Mismatch
99. Compare T[98] with P[0]: Match
100. Compare T[99] with P[0]: Mismatch
101. Compare T[100] with P[0]: Match
102. Compare T[101] with P[0]: Mismatch
103. Compare T[102] with P[0]: Match
104. Compare T[103] with P[0]: Mismatch
4.3 Pattern Matching for Strings

Example 3 for naïve matching

\[
\begin{align*}
T &= \text{abcdef} \quad \text{abcdef} \\
P &= \text{abcdef} \star \\
\text{aaa} &\quad \text{abcdef} \checkmark
\end{align*}
\]
4.3 Pattern Matching for Strings

**Naïve Matching Algorithm: Part one**

```c
int FindPat_1(string S, string P, int startindex) {
    // Countdown a pattern length of positions from the tail of S
    int LastIndex = S.length() - P.length();
    int count = P.length();
    // If startindex (the value of the position that start to match)
    // is too large, matching can not be successful.
    if (LastIndex < startindex)
        return (-1);
    // g is the cursor for S, compare pattern P with the gth position of S,
    // if it fails, continue to loop
    for (int g = startindex; g <= LastIndex; g++) {
        if (P == S.substr(g, count))
            return g;
    }
    // if the for loop is finished, the entire matching fails,
    // and the negative value will be returned
    return (-1);
}
```
4.3 Pattern Matching for Strings

Naïve Matching Algorithm: Part two

```c++
int FindPat_2(string T, string P, int startindex) {
    // countdown a template length of positions from the end of T
    int LastIndex = T.length() - P.length();
    // If startindex (the value of the position that start to match) is too large,
    // matching can not be successful.
    if (LastIndex < startindex) return (-1);
    // i is a cursor pointing to internal characters of T,
    // j is a cursor pointing to internal characters of P.
    int i = startindex, j = 0;
    while (i < T.length() && j < P.length()) // “<=” ?
        if (P[j] == T[i]) {
            i++; j++;
        } else {
            i = i - j + 1; j = 0;
        }
    if (j >= P.length()) // “>” ?
        return (i - j); // if succeeds, return the starting position of T’s substring
    else return -1; // if fails, return negative value
}
```
4.3 Pattern Matching for Strings

Naïve Matching Algorithm Code (simple)

```c
int FindPat_3(string T, string P, int startindex) {
    // g is the cursor of T, compare pattern P with the gth position of S,
    // if fails, continue to loop
    for (int g=startindex; g <= T.length() - P.length(); g++) {
        for (int j=0; ((j<P.length()) && (T[g+j]==P[j])); j++) ;
        if (j == P.length())
            return  g;
    }
    return(-1); // The end of ‘for’, or startingindex is too large, the match fails
}
```
4.3 Pattern Matching for Strings

The original pattern matching algorithm
----- complexity analysis

- $m \leq n$ Assume that the length of the target string $T'$ is $n$, the length of the pattern $P$ is $m$, $m \leq n$
  - In the worst case, each loop is not successful, the number of comparisons will be $(n-m+1)$.
  - The time that every “same matching” takes is time of the comparision of $P$ and $T$ character by character. In worst case, a total of $m$ times.

Thus, the worst time for entire algorithm is:

$O(m \cdot n)$
Naïve Matching Algorithm : worst case

- Compare pattern with every substring of target String that has a length of \( n \)
  - Target String like: \( a^{n-1}X \)
  - Pattern like \( a^{m-1}b \)

- The total number of comparisons :
  - \( m(n - m + 1) \)

- Time complexity :
  - \( O(mn) \)
4.3 Pattern Matching for Strings

Naïve Matching Algorithm: Best case-find patterns

- Find patterns on the m-th position in front of the target string, assign m=5
  
  AAAAA AAAAAAAAAAAAAAAAAA

  Five times comparison

- The total number of comparisons: m
- Time complexity: O(m)
Naïve Matching Algorithm:
Best case—don’t find patterns

- Always mismatch on the first character
- The total number of comparisons
  - $n - m + 1$
- Time complexity:
  - $O(n)$
4.3 Pattern Matching for Strings

Thinking : redundant operation of naïve algorithm

- The reason why naïve algorithm is slow is the redundant operation.
- e.g.,
  - From 1 we can know : \( p_5 \neq t_5 \), \( p_0 = t_0, p_1 = t_1 \), at the same time, from \( p_0 \neq p_1 \) we can get \( p_0 \neq t_2 \)
  
  So when you moves P one place right, the 2\textsuperscript{nd} comparison is sure to be unequal.

  The comparison is redundant

- How many positions should you move P right to eliminate the redundancy operation without losing “matched-string”? 

\[
\begin{align*}
\text{T} & \quad \text{a b a c a a b a c a a b a c a a} \\
\text{P} & \quad \text{a b a c a b} \\
1) & \quad p_5 \neq T_5 \quad \text{P move one place right} \\
\text{T} & \quad \text{a b a c a a b a c a a b a c a a} \\
\text{P} & \quad \text{a b a c a b} \\
2) & \quad p_0 \neq T_1 \quad \text{P move one place right} \\
\text{T} & \quad \text{a b a c a a b a c a a b a c a a} \\
\text{P} & \quad \text{a b a c a b} \\
3) & \quad p_1 \neq T_3 \quad \text{P move one place right} \\
\text{T} & \quad \text{a b a c a a b a c a a b a c a a} \\
\text{P} & \quad \text{a b a c a b} \\
\text{......} & \\
\end{align*}
\]
Outline

• The Basic Concept of Strings
• The Storage Structure of Strings
• The Implementation of Strings’ Operations
• Pattern Matching for Strings
  – Naïve algorithm
  – KMP algorithm
4.3 Pattern Matching for Strings

Match without Backtracking

- In matching process, once $p_j$ is not equal to $t_i$, that’s:
  $P\text{.substr}(1,j-1) == T\text{.substr}(i-j+1,j-1)$
  
  But $p_j \neq t_i$

  - Which character $p_k$ should be used to compare with $t_i$ in $p$?
  - Determine the number of right-moving of digits
  - It is clear that $k < j$, and when $j$ changes, $k$ will change too

- Knuth-Morrit-Pratt (KMP) algorithm

  - The value of $k$ only depends on pattern $P$ itself, it doesn’t have relations with target string $T$
4.3 Pattern Matching for Strings

KMP algorithm

\[ T = \text{abcdefabcdef}\]

\[ P = \text{abcdef}\]

We have

\[ t_{i-j} t_{i-j+1} t_{i-j+2} \ldots t_{i-2} t_{i-1} t_i \ldots t_{n-1} = p_0 p_1 p_2 \ldots p_{j-1} \] (1)

**naive for next trip**

\[ p_0 p_1 \ldots p_{j-2} \neq p_1 p_2 \ldots p_{j-1} \] (2)

You can immediately conclude:

\[ p_0 p_1 \ldots p_{j-2} \neq t_{i-j+1} t_{i-j+2} \ldots t_{i-1} \]

(naive matching) Next trip will not match, jump over

\[ p_0 p_1 \ldots p_{j-2} p_{j-1} \]
4.3 Pattern Matching for Strings

It’s same, if \( p_0 p_1 \ldots p_{j-3} \neq p_2 p_3 \ldots p_{j-1} \)

It doesn’t match in the next step, because

\[
p_0 p_1 \ldots p_{j-3} \neq t_{i-j+2} t_{i-j+3} \ldots t_{i-1}
\]

Until, “\( k \)” appears (the length of head and tail string), it makes

\[
p_0 p_1 \ldots p_{k-1} \neq p_{j-k-1} p_{j-k} \ldots p_{j-1}
\]

and

\[
p_0 p_1 \ldots p_{k-1} = p_{j-k} p_{j-k+1} \ldots p_{j-1}
\]

Pattern moves \( j-k \) bits right

\[
\begin{array}{c|c|c|c}
  \hline
  t_{i-k} & t_{i-k+1} & \ldots & t_{i-1} \\
  \hline
  p_{j-k} & p_{j-k+1} & \ldots & p_{j-1} \\
  \hline
  p_0 & p_1 & \ldots & p_{k-1} & p_k \\
  \hline
\end{array}
\]

\[
\text{So } p_0 p_1 \ldots p_{k-1} = t_{i-k} t_{i-k+1} \ldots t_{i-1}
\]
4.3 Pattern Matching for Strings

String feature vector: \( N \)

Assume that \( P \) is consisting of \( m \) chars, noted as
\[
P = p_0 p_1 p_2 p_3 \ldots p_{m-1}
\]

Use Feature Vector \( N \) to represent the distribution characteristic of pattern \( P \), which is consisting of \( m \) feature numbers \( n_0 \ldots n_{m-1} \), noted as
\[
N = n_0 n_1 n_2 n_3 \ldots n_{m-1}
\]

\( N \) is also called as the next array, each element \( n_j \) corresponds to next[j]
4.3 Pattern Matching for Strings

Feature vector for String N: Constructor

- The feature number of the j-th position of P is $n_j$
- The longest head and tail string is $k$
  - Head string: $p_0 \ p_1 \ \ldots \ p_{k-2} \ p_{k-1}$
  - Tail String: $p_{j-k} \ p_{j-k+1} \ \ldots \ p_{j-2} \ p_{j-1}$

$$\text{next}[j] = \begin{cases} -1, & j = 0 \\ \max\{k: 0 < k < j \ \& \ P[0...k-1] = P[j-k...j-1]\}, & \text{If } K \text{ exists} \\ 0, & \text{otherwise} \end{cases}$$
### 4.3 Pattern Matching for Strings

**Strings**

- **Pattern (P)**:
  - `a a a a a b a a a a c`

- **Text (T)**:
  - `a a b a a a a a a a b a a a a a c b`

- **Next Array (N)**:
  - `0 1 2 1 0 1 2 3 4`

- **X**: (should be 3)

**Miss it!**

- **Missed Pattern**: `a a a a a b a a a a c`
### Example for Pattern Matching for Strings

**P** = 
```
0 1 2 3 4 5 6
```
```
 a b a b a b b
```

**N** = 
```
0 1 2 3 4
```
```
-1 0 0 1 2 3 4
```

**T** = 
```
0 1 2 3 4 5 6 7 8 9 10 11 12
```
```
 a b a b a b b a b a b a b b
```

**P** = 
```
0 1 2 3 4 5 6
```
```
 a b a b a b b a b a b a b b
```

**N** = 
```
0 1 2 3 4
```
```
-1 0 0 1 2 3 4
```

**T** = 
```
0 1 2 3 4 5 6 7 8 9 10 11 12
```
```
 a b a b a b b a b a b a b b
```

- **i=6, j=6, N[j]=4**
- **i=8, j=6, N[j]=4**
- **i=10, j=6, j'=4**
### 4.3 Pattern Matching for Strings

#### KMP matching algorithm

```c
int KMPStrMatching(string T, string P, int *N, int start) {
    int j = 0;  // subscript variable of pattern
    int i = start;  // subscript variable of target string
    int pLen = P.length();  // length of pattern
    int tLen = T.length();  // length of target string
    if (tLen - start < pLen) // if the target is shorter than
        // the pattern, matching can not succeed
        return (-1);
    while (j < pLen && i < tLen) { // repeat comparisons to match
        if (j == -1 || T[i] == P[j])
            i++, j++;
        else j = N[j];
    }
    if (j >= pLen)
        return (i-pLen);  // be careful with the subscript
    else return (-1);
}
```
Algorithm Framework for Seeking the Feature Value

- Feature value $n_j$ ( $j > 0$, $0 \leq n_{j+1} \leq j$ ) is recursively defined, defined as follows:
  1. $n_0 = -1$, for $n_{j+1}$ with $j > 0$, assume that the feature value of the previous position is $n_j$, let $k = n_j$;
  2. When $k \geq 0$ and $p_j \neq p_k$, let $k = n_k$; let Step 2 loop until the condition is not satisfied.
  3. $n_{j+1} = k + 1$ ; // $k = -1$ or $p_j = p_k$
4.3 Pattern Matching for Strings

Feature vector of String: $N$ — non-Optimized version

```cpp
int findNext(string P) {
    int j, k;
    int m = P.length(); // m is the length of pattern P
    assert( m > 0); // if m=0, exit
    int *next = new int[m]; // open up an integer array in dynamic storage area.
    assert( next != 0); // if opening up integer array fails, exit
    next[0] = -1;
    j = 0;  k = -1;
    while (j < m-1) {
        while (k >= 0 && P[k] != P[j]) // ff not equal, use kmp to look for head and
tail substring
            k = next[k]; // k recursively looking forward
        j++;   k++;  next[j] = k;
    }
    return next;
}
```
4.3 Pattern Matching for Strings

**Seeking feature vector N**

\[
N = \begin{bmatrix}
-1 & 0 & 1 & 2 & 3 & 0 & 1 & 2 & 3 & 4 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9
\end{bmatrix}
\]

\[
P = \begin{bmatrix}
a & a & a & a & b & a & a & a & a & c \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9
\end{bmatrix}
\]

\[j = 9 \quad k = 0\]
Pattern moves j-k bit right

\[ t_{i:j} \quad t_{i:j+1} \quad t_{i:j+2} \quad \ldots \quad t_{i:k} \quad t_{i:k+1} \quad \ldots \quad t_{i-1} \quad t_i \]

\[ p_0 \quad p_1 \quad p_2 \quad \ldots \quad p_{j-k} \quad p_{j-k+1} \quad \ldots \quad p_{j-1} \quad p_j \]

\[ p_0 \quad p_1 \quad \ldots \quad p_{k-1} \quad p_k \]

\[ p_0 \quad p_1 \ldots p_{k-1} = t_{i-k} \quad t_{i-k+1} \quad \ldots \quad t_{i-1} \]

\[ t_i \neq p_j , \quad p_j = p_k \]
4.3 Pattern Matching for Strings

KMP Matching

<table>
<thead>
<tr>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>K</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Target: a b c a a b a b c

N[1] = 0
N[3] = 0
N[0] = -1
N[6] = 2

P[3] == P[0], P[3] \neq T[4], one more time comparison is redundant
4.3 Pattern Matching for Strings

Feature vector of String: \( N \) — Optimized version

```cpp
int findNext(string P) {
    int j, k;
    int m = P.length(); // m is the length of pattern P
    int *next = new int[m]; // open up an integer array in dynamic storage area.
    next[0] = -1;
    j = 0; k = -1;
    while (j < m-1) {
        while (k >= 0 && P[k] != P[j]) // if not equal, use kmp to look for head and tail substring
            k = next[k]; // k looks forward recursively
        j++; k++;
        if (P[k] == P[j]) // finding value k isn’t affected by the optimization
            next[j] = next[k];
        else next[j] = k; // no optimization if you cancel the “if” judgment
    }
    return next;
}
```
### 4.3 Pattern Matching for Strings

#### Comparison of next arrays

<table>
<thead>
<tr>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>k</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>$p_k == p_j$?</td>
<td>≠</td>
<td>≠</td>
<td>= =</td>
<td>≠</td>
<td>= =</td>
<td>≠</td>
<td>= =</td>
<td>= =</td>
<td></td>
</tr>
<tr>
<td>next[$j$]</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Non-optimized version

Optimized version
4.3 Pattern Matching for Strings

Time Analysis for the KMP algorithm

- The statement “j = N[j];” in the loop will not execute more than n times. Otherwise,
  - Because every time the statement “j = N[j];” is executed, j decreases inevitably (minus at least by one)
  - Only “j++” can increase j
  - Thus, if the statement “j==N[j]” is executed more than n times, j will become smaller than -1. It’s impossible (sometimes j becomes -1, but it will be increased by 1 and becomes 0 immediately)
- The time for constructing the N array is O(m)

Therefore, the time complexity of KMP is O(n+m)
## 4.3 Pattern Matching for Strings

### Summary: Single-matching algorithm

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time efficiency for preprocessing</th>
<th>Time efficiency for matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve matching algorithm</td>
<td>$\Theta(0)$</td>
<td>$\Theta(n \cdot m)$</td>
</tr>
<tr>
<td>KMP</td>
<td>$\Theta(m)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>BM</td>
<td>$\Theta(m)$</td>
<td>Best $(n/m)$, Worst $\Theta(nm)$</td>
</tr>
<tr>
<td>shift-or, shift-and</td>
<td>$\Theta(m +</td>
<td>\Sigma</td>
</tr>
<tr>
<td>Rabin-Karp</td>
<td>$\Theta(m)$</td>
<td>Average $(n+m)$, Worst $\Theta(nm)$</td>
</tr>
<tr>
<td>Finite state automaton</td>
<td>$\Theta(m \cdot</td>
<td>\Sigma</td>
</tr>
</tbody>
</table>
4.3 Pattern Matching for Strings

Different Versions of the Feature Vector

If match fails on the j-th character, let $j = \text{next}[j]$

$$\text{next}[j] = \begin{cases} -1, & \text{If } j == 0 \\ \max\{k: 0 < k < j \land \land P[0...k-1] = P[j-k...j-1] \}, & \text{If } K \text{ exists} \\ 0, & \text{else} \end{cases}$$

If match fails on the j-th character, let $j = \text{next}[j-1]$

$$\text{next}[j] = \begin{cases} 0, & \text{If } j == 0 \\ \max\{k: 0 < k < j \land \land P[0...k] = P[j-k...j] \}, & \text{If } K \text{ exists} \\ 0, & \text{else} \end{cases}$$
Reference materials

- Pattern Matching Pointer

- EXACT STRING MATCHING ALGORITHMS
  - http://www-igm.univ-mlv.fr/~lecroq/string/
  - Description and complexity analysis of pattern matching for strings and the C source code
Data Structures and Algorithms

Thanks

the National Elaborate Course (Only available for IPs in China)
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

Ming Zhang, Tengjiao Wang and Haiyan Zhao
Higher Education Press, 2008.6 (awarded as the "Eleventh Five-Year" national planning textbook)