数据结构
Data Structures

Chapter 7 Strings

Prof. Yitian Shao
School of Computer Science and Technology

Strings

Course Overview

- Strings
  - Review of C++ syntax
- Pattern searching algorithms
- Brute force searching
- KMP algorithm
  - Preprocess the pattern (LPS)
  - Searching the text

Strings

- A **string** is a **sequence of characters**
- There are two types of strings in C++: C strings (**char arrays**) and C++ strings (**string objects**)
- A string literal such as “hello world" is a C string
- Converting between the two types:

```
string my_str("text"); // C string to C++ string
char* my_c = my_str.c_str(); // C++ string to C string
```

Previously in High-level Language Programming

- C++ strings defined by a class
- The string data type is not built into C++

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

int main()
{
    string password = " secret " ;
    string user_input ;
    cout << " Enter Password: " ;
    cin >> user_input ;
    if ( password == user_input )
        cout << " Correct password. Welcome to the system ... " << endl ;
    else
        cout << " Invalid password " << endl ;
    return 0;

}
```

- C++ strings are compared using **==, !=, etc** instead of *strcmp*() in C-strings

Previously in High-level Language Programming

- **C++ string concatenation**

```
string str1 = “Hello”, str2 = “World”;

str3 = str1 + str2;          → str3  “HelloWorld”

str1.append(str3)            → str1  “HelloHelloWorld”
```

Previously in High-level Language Programming

- **C++ string swap**

```
string str1 = “Hello”, str2 = “World”;


str1.swap(str2);  →  str1 “World”  str2 “Hello”
```

Previously in High-level Language Programming

- **C++ Character classification**
- The following functions return a true (non-zero integer) value or a false (zero integer) value depending on whether or not the character belongs to a particular set of characters.
- Covert the case of a character: **tolower**() and **toupper**()

Function	Character set
isalnum	Alphanumeric character: A-Z, a-z, 0-9
isalpha	Alphabetic character: A-Z, a-z
isascii	ASCII character: ASCII codes 0-127
iscntrl	Control character: ASCII codes 0-31 or 127
isdigit	Decimal digit: 0-9
isgraph	Any printable character other than a space
islower	Lowercase letter: a-z
isprint	Any printable character, including a space
ispunct	Any punctuation character
isspace	Whitespace character: \t,\v,\f,\r,\n or space ASCII codes 9-13 or 32
isupper	Uppercase letter: A-Z
isxdigit	Hexadecimal digit: 0-9 and A-F

Function	Purpose
tolower	Converts an uppercase character to lowercase.
toupper	Converts a lowercase character to uppercase.

Previously in High-level Language Programming

- **C++ string pattern searching**

```
              012345678
string s = "Welcome to Data Structure!";
string sub = "to";
cout << s.find(sub); // (Print out) 8
string sub2 = "hello";
if (s.find(sub2) != string::npos)
    cout << “Not found”;
```

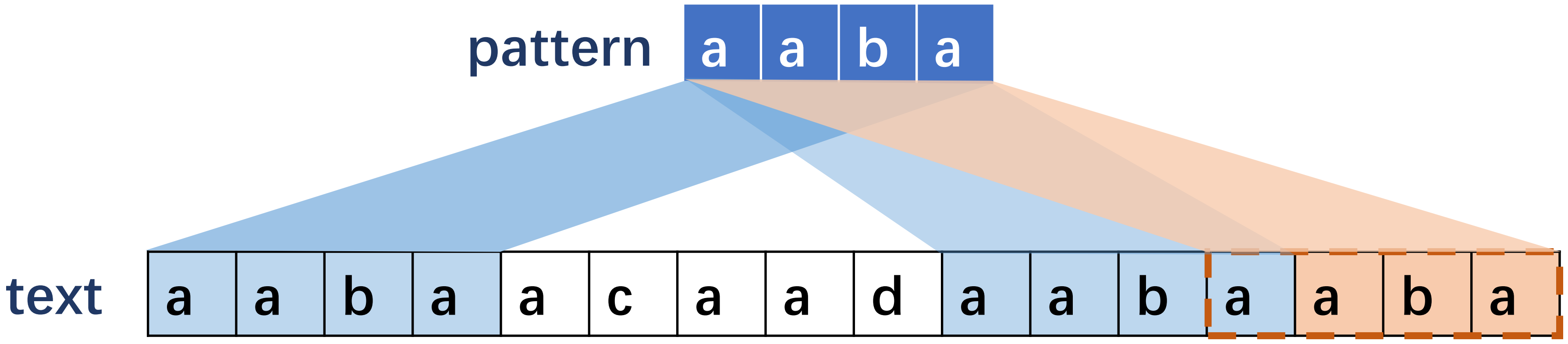
`string::npos` is returned if no substring can be found

Pattern Searching Algorithm for Strings

- How to implement the `find()` function?
- How to find all matched patterns in the text?

Pattern Searching Example

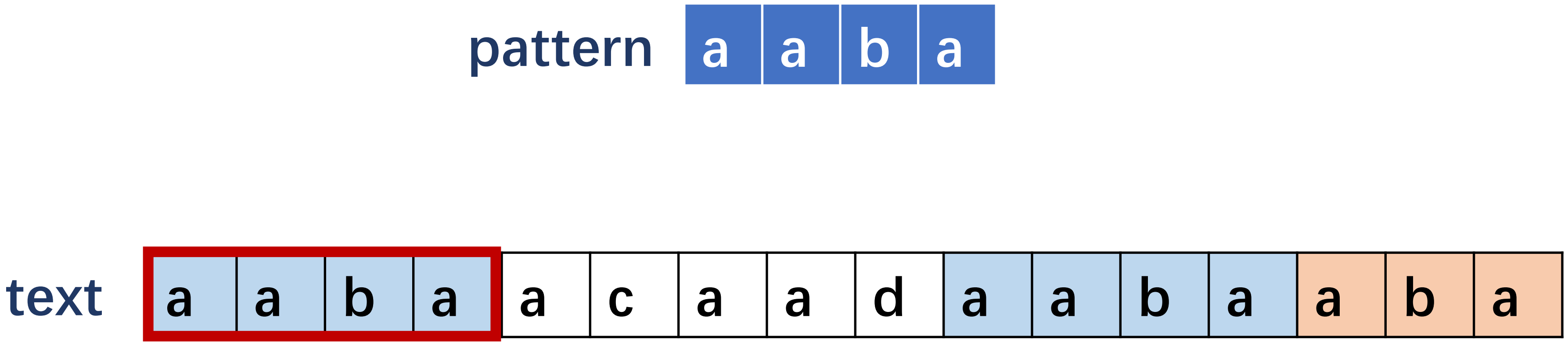
- string text = “aabaacaadaabaaba”, pattern = “aaba”;



- Answer: [0, 9, 12]

Brute Force Searching

- Slide the pattern over text **one by one** and check for a match. If a match is found, then slide by one character again to check for subsequent matches



In-Class Exercise

- Implement the brute force searching using C++

```
void search(string& pat, string& txt) {

    for(        ){  // Slide pattern window by 1 step repetitively



        for(        ){  // Check for pattern match for each i



        if(        ) {  // If pattern matches at index i
            cout << "Pattern found at index " << i << endl;
        }
    }
}
```

In-Class Exercise: Solution

- Implement the brute force searching using C++

```
void search(string& pat, string& txt) {
    int M = pat.size();
    int N = txt.size();

    for(int i = 0; i <= N - M; i++) { // Slide pattern window by 1 step repetitively

        for(        ){ // Check for pattern match for each i




        if(        ) { // If pattern matches at index i
            cout << "Pattern found at index " << i << endl;
        }
    }
}
```

In-Class Exercise: Solution

- Implement the brute force searching using C++

```
void search(string& pat, string& txt) {
    int M = pat.size();
    int N = txt.size();

    for(int i = 0; i <= N - M; i++) { // Slide pattern window by 1 step repetitively
        int j;
        for(j = 0; j < M; j++) { // Check for pattern match for each i
            if (txt[i + j] != pat[j]) {
                break;
            }
        }
        if(       ) { // If pattern matches at index i

        }
    }
}
```

In-Class Exercise: Solution

- Implement the brute force searching using C++

```
void search(string& pat, string& txt) {
    int M = pat.size();
    int N = txt.size();

    for(int i = 0; i <= N - M; i++) { // Slide pattern window by 1 step repetitively
        int j;
        for(j = 0; j < M; j++) { // Check for pattern match for each i
            if (txt[i + j] != pat[j]) {
                break;
            }
        }
        if(j == M) { // If pattern matches at index i
            cout << "Pattern found at index " << i << endl;
        }
    }
}
```

KMP Algorithm

- **Knuth-Morris-Pratt (KMP)** is an efficient string-matching algorithm developed by Donald **Knuth**, James H. **Morris** and Vaughan **Pratt** in 1977, to **find a specific pattern in a given string**.
- To find the smaller string (termed as the "**pattern**") inside a larger string (termed as the "**text**").
- Use a **Longest Prefix Suffix** (**LPS**) array that captures the longest prefix which is also a suffix for every substring.

lps[i] = MAXIMUM(j), where j < pat.size() **such that**
pat.substr(0, j) == pat.substr(i-j+1, j)

KMP Algorithm

- Two main steps of the KMP algorithm:
  - Step 1 - Preprocessing the Pattern: Before searching, KMP creates the **LPS array** based on the pattern that helps **determine how much of the pattern has already been matched**.
  - Step 2 - Searching the Text: Using the LPS array, KMP can quickly **skip unnecessary comparisons** (skip over parts of the text where it's certain the pattern can't match), making the search more efficient.

KMP Algorithm: Step 1 – Preprocess Pattern

- The size of the **LPS array** is same as **pattern** length

  vector<int> lps(pat.size())

- **lps[i] stores j, the length of the longest prefix of pat[0..i] and simultaneously a suffix of pat[0..i], with j < pat.size()**

KMP Algorithm: Step 1 – Preprocess Pattern

- Examples of LPS array construction

Pattern	**LPS array**
pat = "AAAA";	{0, 1, 2, 3}
pat = "ABCDE";	{0, 0, 0, 0, 0}
pat = "AABAACAABAA";	{0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5}
pat = "AAACAAAAAC";	{0, 1, 2, 0, 1, 2, 3, 3, 3, 4}
pat = "AAABAAA";	{0, 1, 2, 0, 1, 2, 3}

lps[i] = MAXIMUM(j), where j < pat.size() **such that**
pat.substr(0, j) == pat.substr(i-j+1, j)

KMP Algorithm: Step 1 – Preprocess Pattern

```
void constructLPS(string &pat, vector<int> &lps) {
    int j = 0; // j stores the longest prefix and suffix of pat[0...i]
    lps[0] = 0; // lps[0] is always 0
    int i = 1;
    while (i < pat.length()) {
        if (pat[i] == pat[j]) { // If characters match, increment the size of lps
            j++;
            lps[i] = j;
            i++; // Note that once j increased, i also increase
        }
        else if (j > 0) { // Mismatch between pat[i] and pat[j] but previous j > 0
            j = lps[j - 1]; // Update j to avoid redundant comparisons
        }
        else { // Mismatch but previous j = 0
            lps[i] = 0; // If no matching prefix found, set lps[i] to 0
            i++;
        }
    }
}
```

AABAACAABAA	{0, 1, 0, 1, j?, ?, ?, ?, ?, ?, ?}

KMP Algorithm: Step 2 – Search Text

- Using `i` and `j` indexing text and pattern: `txt[i]` and `pat[j]`
- When match (`txt[i]=pat[j]`), increment both indices and continue the comparison
- If no match, reset the `j` to the last value from the LPS array, **because that portion of the pattern has already been matched with the text string**

```
txt[3] == pat[3], match!
then make i = 4, j = lps[3] = 1
```

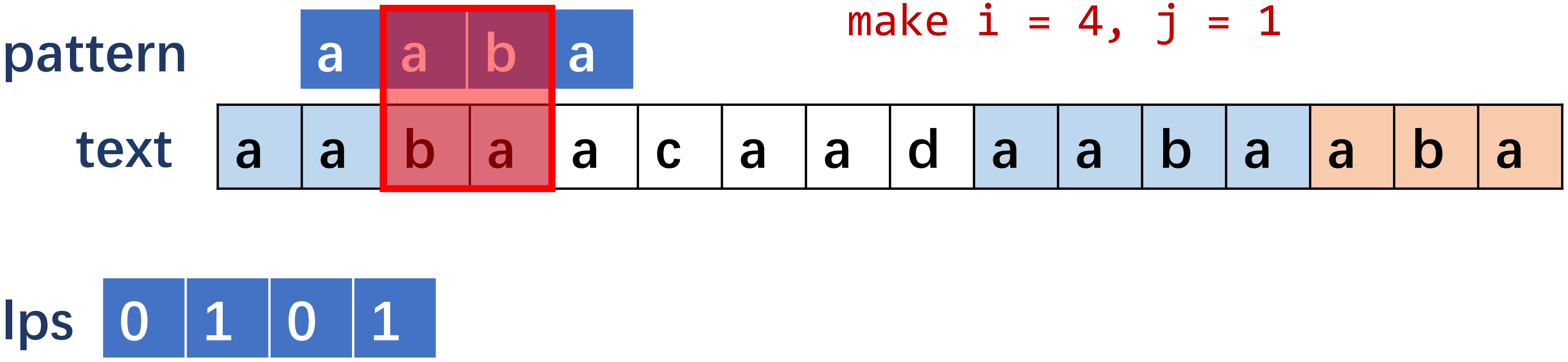
pattern	a	a	b	a												
text	a	a	b	a	a	c	a	a	d	a	a	b	a	a	b	a

lps	0	1	0	1

KMP Algorithm: Step 2 – Search Text

redundant comparison, since lps[3] != 3

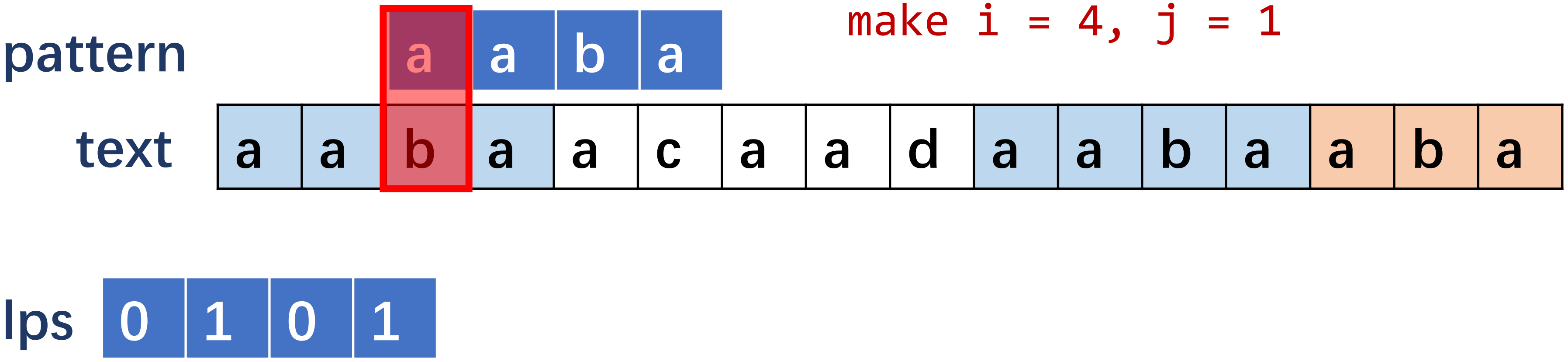
make i = 4, j = 1



KMP Algorithm: Step 2 – Search Text

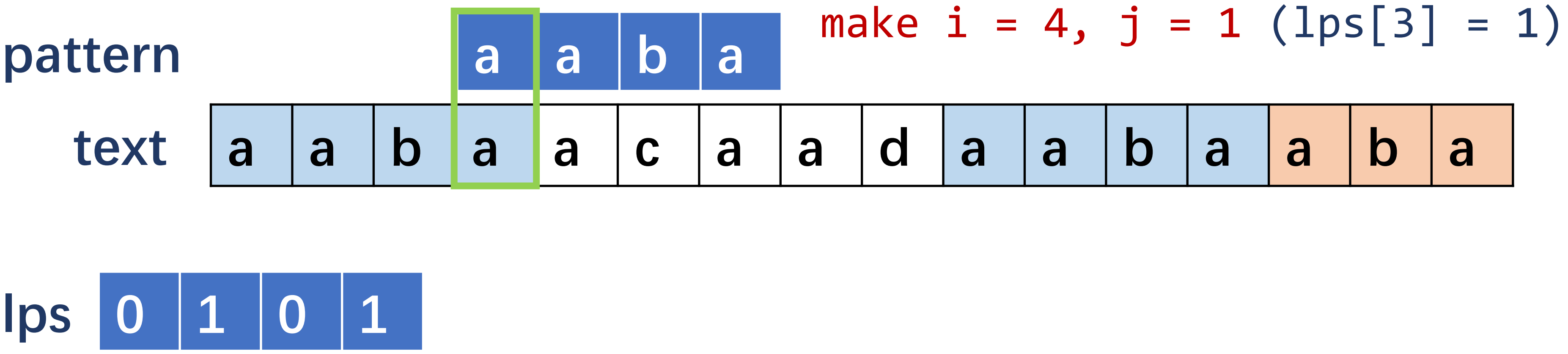
`redundant comparison, since lps[3] != 2`

`make i = 4, j = 1`

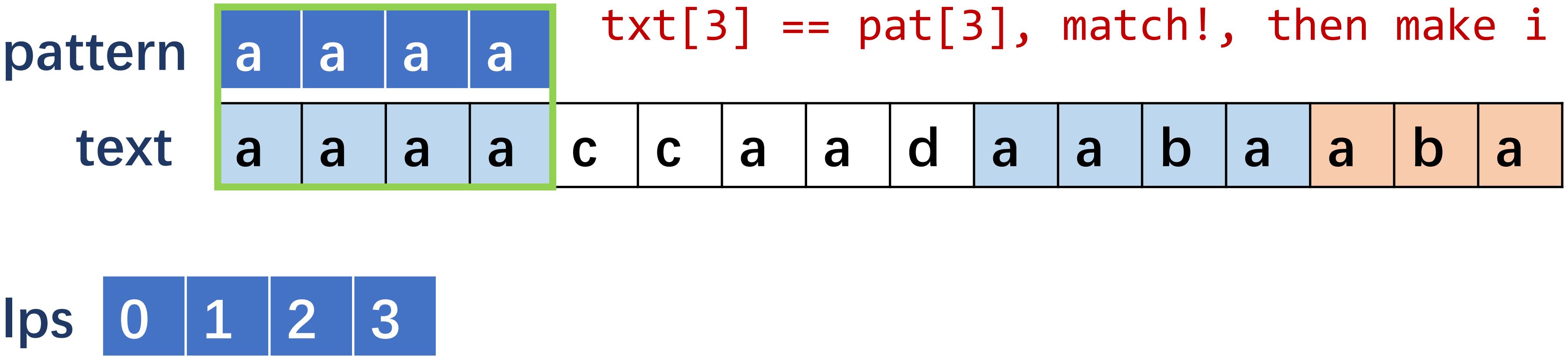


KMP Algorithm: Step 2 – Search Text

- Using `i` and `j` indexing text and pattern: `txt[i]` and `pat[j]`
- When match (`txt[i]=pat[j]`), increment both indices and continue the comparison
- If no match, reset the `j` to the last value from the LPS array, **because that portion of the pattern has already been matched with the text string**

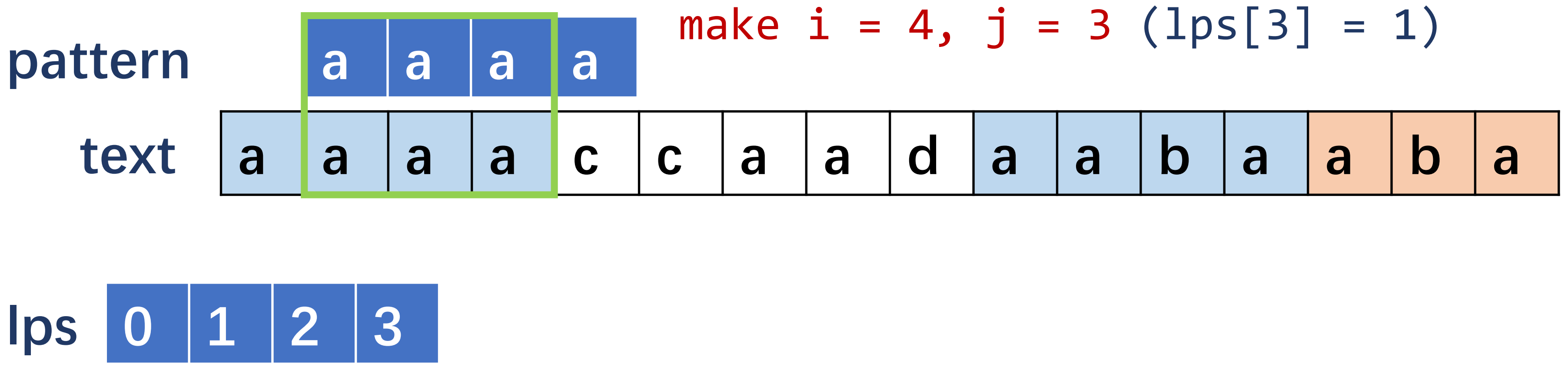


KMP Algorithm: Step 2 – Search Text



txt[3] == pat[3], match!, then make i = 4, j = 3

KMP Algorithm: Step 2 – Search Text



KMP Step 2

```
vector<int> search(string &pat, string &txt) {
    int m = pat.length(), n = txt.length();
    vector<int> lps(m);
    vector<int> res;
    constructLps(pat, lps);
    int i = 0, j = 0; // Index i and j, for traversing the text and pattern
    while (i < n) { // Iterate through the entire text using index i
        if (txt[i] == pat[j]) {// If characters match, move both indices forward
            i++;
            j++;
            if (j == m) { // If the entire pattern is matched
                res.push_back(i - j); // Store the start index in result
                j = lps[j - 1]; // Use LPS of previous index to skip unnecessary comparisons
            }
        }
        else { // If there is a mismatch between txt[i] and pat[j]
            if (j != 0)
                j = lps[j - 1]; // Use previous lps value to avoid redundant comparisons
            else
                i++;
        }
    }
    return res;
}
```

pattern	a	a	b	a

lps	0	1	0	1

text	a	a	b	a	a	c	a	a	d	a	a	b	a	a	b	a

KMP Algorithm: Key Points

- LPS array can tell **how much of the pattern has already been matched**
- Skip over parts of the text where it's certain the pattern can't match

Exercise 7.1

- Complete [LeetCode 3248]

3248. Snake in Matrix

Easy | Topics | Companies | Hint

There is a snake in an `n x n` matrix `grid` and can move in **four possible directions**. Each cell in the `grid` is identified by the position: `grid[i][j] = (i * n) + j`.

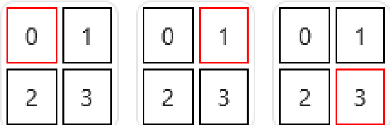
The snake starts at cell 0 and follows a sequence of commands.

You are given an integer `n` representing the size of the `grid` and an array of strings `commands` where each `command[i]` is either `"UP"`, `"RIGHT"`, `"DOWN"`, and `"LEFT"`. It's guaranteed that the snake will remain within the `grid` boundaries throughout its movement.

Return the position of the final cell where the snake ends up after executing `commands`.

Example 1:

> **Input:** `n = 2, commands = ["RIGHT","DOWN"]`
>
> **Output:** 3
>
> **Explanation:**



Exercise 7.2

- Complete [LeetCode 682]

682. Baseball Game

Easy | Topics | Companies

You are keeping the scores for a baseball game with strange rules. At the beginning of the game, you start with an empty record.

You are given a list of strings `operations`, where `operations[i]` is the i^{th} operation you must apply to the record and is one of the following:

- An integer `x`.
  - Record a new score of `x`.
- `'+'`.
  - Record a new score that is the sum of the previous two scores.
- `'D'`.
  - Record a new score that is the double of the previous score.
- `'C'`.
  - Invalidate the previous score, removing it from the record.

Return *the sum of all the scores on the record after applying all the operations*.

The test cases are generated such that the answer and all intermediate calculations fit in a **32-bit** integer and that all operations are valid.

Exercise 7.3

- Complete [LeetCode 1684]

1684. Count the Number of Consistent Strings

Easy Topics Companies Hint

You are given a string `allowed` consisting of **distinct** characters and an array of strings `words`. A string is **consistent** if all characters in the string appear in the string `allowed`.

Return *the number of **consistent** strings in the array* `words`.

Example 1:

```
Input: allowed = "ab", words = ["ad","bd","aaab","baa","badab"]
Output: 2
Explanation: Strings "aaab" and "baa" are consistent since they only contain characters 'a' and 'b'.
```

Example 2:

```
Input: allowed = "abc", words = ["a","b","c","ab","ac","bc","abc"]
Output: 7
Explanation: All strings are consistent.
```

Exercise 7.4

- Complete LeetCode 459

459. Repeated Substring Pattern

Easy Topics Companies

Given a string `s`, check if it can be constructed by taking a substring of it and appending multiple copies of the substring together.

Example 1:

```
Input: s = "abab"
Output: true
Explanation: It is the substring "ab" twice.
```

Example 2:

```
Input: s = "aba"
Output: false
```

Exercise 7.5

- Complete [LeetCode 1392]

1392. Longest Happy Prefix

Hard | Topics | Companies | Hint

A string is called a **happy prefix** if is a **non-empty** prefix which is also a suffix (excluding itself).

Given a string `s`, return *the* ***longest happy prefix*** *of* `s`. Return an empty string `""` if no such prefix exists.

Example 1:

```
Input: s = "level"
Output: "l"
Explanation: s contains 4 prefix excluding itself ("l", "le", "lev", "leve"), and suffix ("l", "el", "vel", "evel").
The largest prefix which is also suffix is given by "l".
```

Example 2:

```
Input: s = "ababab"
Output: "abab"
Explanation: "abab" is the largest prefix which is also suffix. They can overlap in the original string.
```