**HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG**

**KHOA CÔNG NGHỆ THÔNG TIN 1**

Icon

Description automatically generated

**CHUYÊN ĐỀ CÔNG NGHỆ PHẦN MỀM**

|  |
| --- |
| **CHỦ ĐỀ: PATTERN SEARCHING** |
| **Giảng viên hướng dẫn: thầy Nguyễn Duy Phương** |
| **Sinh viên : Vũ Đình Công** |
| **Mã sinh viên: B19DCCN074** |
| **Số điện thoại: 0865794296**  **Nhóm : 7** |
| **Gmail:** [**Cong080201@gmail.com**](mailto:Cong080201@gmail.com) |

**LỜI CẢM ƠN**

Em xin gửi lời cảm ơn chân thành đến Thầy Nguyễn Duy Phương- giảng viên môn học Chuyên đề Công nghệ phần mềm, Học viện Công nghệ Bưu chính Viễn thông, vì sự hỗ trợ, sự dẫn dắt và kiến thức quý báu mà Thầy đã chia sẻ trong suốt quá trình em học môn này.

Em xin được bày tỏ lòng kính trọng và tôn trọng đối với Thầy, người đã chia sẻ kiến thức chuyên môn và kinh nghiệm thực tế trong lĩnh vực công nghệ thông tin. Nhờ sự dạy dỗ và hướng dẫn tận tâm của Thầy, em đã được khám phá, hiểu rõ hơn về các khía cạnh quan trọng trong ngành và trang bị những kỹ năng cần thiết để phát triển bản thân. Em cũng rất biết ơn vì những phản hồi và sự hỗ trợ từ Thầy trong quá trình thực hiện dự án. Thầy đã luôn sẵn lòng trả lời các câu hỏi của em, giải đáp những thắc mắc và đưa ra những đề xuất cải tiến. Sự tận tâm và tâm huyết của Thầy đã thực sự truyền cảm hứng cho em trong việc học tập và nghiên cứu.

Cuối cùng, em xin kính chúc Thầy sức khỏe, hạnh phúc và thành công trong sự nghiệp giảng dạy. Em rất biết ơn và tự hào vì được học từ Thầy. Mong rằng em sẽ tiếp tục được học hỏi và phát triển dưới sự hướng dẫn của Thầy trong những khoá học tương lai.

Em xin chân thành cảm ơn!

Trân trọng

Vũ Đình Công

**Mục lục**

[**I.** **Tìm kiếm mẫu từ trái sang phải** 4](#_Toc136204961)

1. [Thuật toán Forward Dawg Matching 4](#_Toc136204962)
2. [Thuật toán Turbo Reverse Factor 6](#_Toc136204963)
3. [Thuật toán Knuth-Morris-Pratt 8](#_Toc136204964)
4. [Thuật toán Shift Or 10](#_Toc136204965)
5. [Thuật toán Simon 12](#_Toc136204966)
6. [Thuật toán Apostolico-Crochemore 15](#_Toc136204967)
7. [Thuật toán Karp-Rabin 17](#_Toc136204968)

[**II.** **Tìm kiếm mẫu từ phải sang trái** 19](#_Toc136204969)

1. [Thuật toán Turbo BM 19](#_Toc136204970)
2. [Thuật toán Zhu-Takaoka 22](#_Toc136204971)
3. [Thuật toán Berry-Ravindran 24](#_Toc136204972)
4. [Thuật toán Quick Search 26](#_Toc136204973)
5. [Thuật toán Reverse Colussi 28](#_Toc136204974)
6. [Thuật toán Tuned Boyer-Moore 30](#_Toc136204975)
7. [Thuật toán Reverse Factor algorithm 33](#_Toc136204976)

[**III.** **Tìm kiếm mẫu từ vị trí xác định** 34](#_Toc136204977)

1. [Thuật toán KMP Skip Search 34](#_Toc136204978)
2. [Thuật toán Alpha Skip Search 36](#_Toc136204979)
3. [Thuật toán Maximal Shift 39](#_Toc136204980)
4. [Thuật toán Optimal Mismatch 41](#_Toc136204981)
5. [Thuật toán Skip Search 43](#_Toc136204982)
6. [Thuật toán Two way 45](#_Toc136204983)
7. [Thuật toán Galil-Giancarlo 48](#_Toc136204984)
8. [Thuật toán Colussi 50](#_Toc136204985)

[**IV.** **Tìm kiếm mẫu từ vị trí bất kì** 52](#_Toc136204986)

1. [Thuật toán Smith 52](#_Toc136204987)
2. [Thuật toán Raita 54](#_Toc136204988)
3. [Thuật toán Apostolico-Giancarlo 57](#_Toc136204989)
4. [Thuật toán Horspool 59](#_Toc136204990)

[**TỔNG KẾT** 62](#_Toc136204991)

# **Tìm kiếm mẫu từ trái sang phải**

## Thuật toán Forward Dawg Matching

Forward Dawg Matching là một thuật toán tìm kiếm chuỗi con trong một chuỗi đầu vào. Thuật toán này kết hợp giữa cấu trúc DAWG (Directed Acyclic Word Graph) và kỹ thuật tìm kiếm tiến (forward search). DAWG là một cấu trúc dữ liệu đồ thị hướng không có chu trình, được sử dụng để lưu trữ tập hợp các từ hoặc chuỗi con của một ngôn ngữ.

Thuật toán Forward Dawg Matching sử dụng DAWG để xác định các đường đi từng bước trong chuỗi đầu vào, từ đó tìm kiếm chuỗi con. Quá trình tìm kiếm được thực hiện bằng cách di chuyển theo các cạnh của DAWG, kiểm tra ký tự tại mỗi bước và so sánh với chuỗi con cần tìm. Khi tìm thấy chuỗi con, thuật toán trả về vị trí đầu tiên của chuỗi con trong chuỗi đầu vào hoặc một danh sách các vị trí nếu có nhiều kết quả.

Đánh giá độ phức tạp của thuật toán:

* Độ phức tạp trong trường hợp tốt nhất: O(m), m là độ dài của chuỗi con cần tìm.
* Độ phức tạp trong trường hợp trung bình: O(n), n là độ dài của chuỗi đầu vào.
* Độ phức tạp trong trường hợp xấu nhất: O(n), n là độ dài của chuỗi đầu vào.

Kiểm nghiệm thuật toán Forward Dawg Matching:

Chuỗi đầu vào: "ababcababcabc"

Chuỗi con: "abc"

Kết quả kỳ vọng: Tìm thấy chuỗi con tại vị trí 2.

Chuỗi đầu vào: "abababababab"

Chuỗi con: "aba"

Kết quả kỳ vọng: Tìm thấy chuỗi con tại các vị trí 1, 3, 5, 7 và 9.

Chuỗi đầu vào: "abcdefg"

Chuỗi con: "xyz"

Kết quả kỳ vọng: Không tìm thấy chuỗi con trong chuỗi đầu vào.

**#include <iostream>**

**#include <string>**

**#include <vector>**

**using namespace std;**

**vector<int> forwardDawgMatching(const string& text, const string& pattern) {**

**vector<int> positions;**

**int n = text.length();**

**int m = pattern.length();**

**int i = 0;**

**while (i <= n - m) {**

**int j = 0;**

**while (j < m && pattern[j] == text[i + j]) {**

**j++;**

**}**

**if (j == m) {**

**positions.push\_back(i); // Tìm thấy chuỗi con, lưu vị trí**

**}**

**i++;**

**}**

**return positions;**

**}**

**int main() {**

**string text = "ababcababcabc";**

**string pattern = "abc";**

**vector<int> positions = forwardDawgMatching(text, pattern);**

**if (!positions.empty()) {**

**cout << "Pattern found at positions: ";**

**for (int i : positions) {**

**cout << i << " ";**

**}**

**cout << endl;**

**} else {**

**cout << "Pattern not found in the text." << endl;**

**}**

**return 0;**

**}**

## Thuật toán Turbo Reverse Factor

Thuật toán Turbo Reverse Factor là một thuật toán tìm kiếm mẫu trong chuỗi (pattern matching) dựa trên kỹ thuật xây dựng và sử dụng bảng Reverse Factor để tăng tốc quá trình tìm kiếm. Thuật toán này kết hợp giữa các kỹ thuật của thuật toán Boyer-Moore và thuật toán Reverse Factor để cải thiện hiệu suất tìm kiếm.

Sơ lược về thuật toán Turbo Reverse Factor:

* Tiền xử lý: Xây dựng bảng Reverse Factor cho mẫu. Bảng này chứa thông tin về sự xuất hiện của các yếu tố đảo ngược trong mẫu.
* Tìm kiếm: Bắt đầu từ vị trí cuối cùng của mẫu trong văn bản và di chuyển từ phải sang trái.
* So sánh: So sánh các ký tự từ phải sang trái của mẫu với văn bản.
* Nếu các ký tự khớp, tiếp tục so sánh với các ký tự tiếp theo theo chiều từ phải sang trái.
* Nếu không khớp, sử dụng thông tin từ bảng Reverse Factor để di chuyển tới vị trí tiếp theo có thể chứa một phần của mẫu.
* Lặp lại quá trình cho đến khi tìm thấy mẫu hoặc hết văn bản.

Đánh giá độ phức tạp của thuật toán:

* Độ phức tạp trong trường hợp tốt nhất và trung bình là O(n/m), trong đó n là độ dài của chuỗi văn bản và m là độ dài của mẫu. Trong trường hợp này, thuật toán chỉ cần một lần so sánh tại mỗi vị trí.
* Độ phức tạp trong trường hợp xấu nhất là O(n \* m), với n là độ dài của chuỗi văn bản và m là độ dài của mẫu. Trong trường hợp xấu nhất, tất cả các ký tự trong văn bản đều khớp với ký tự cuối cùng của mẫu, khiến thuật toán phải so sánh tại mỗi vị trí.

Kiểm nghiệm thuật toán:

Text: "ABABABA"

Pattern: "ABA"

Kết quả: Tìm thấy mẫu tại các vị trí 0, 2, 4.

Text: "ABCDEF"

Pattern: "XYZ"

Kết quả: Không tìm thấy mẫu trong văn bản.

Text: "AAAAA"

Pattern: "AA"

Kết quả: Tìm thấy mẫu tại các vị trí 0, 1, 2, 3.

Lập trình theo thuật toán:

**#include <iostream>**

**#include <string>**

**#include <vector>**

**std::vector<int> turboReverseFactor(const std::string& text, const std::string& pattern) {**

**std::vector<int> occurrences;**

**int n = text.length();**

**int m = pattern.length();**

**// Xây dựng bảng Reverse Factor cho mẫu**

**std::vector<int> reverseFactor(m, 0);**

**for (int i = 0; i < m; i++) {**

**for (int j = 0; j < m - i - 1; j++) {**

**if (pattern[i + j + 1] == pattern[j]) {**

**reverseFactor[i]++;**

**} else {**

**break;**

**}**

**}**

**}**

**int i = m - 1;**

**while (i < n) {**

**int j = m - 1;**

**while (j >= 0 && text[i] == pattern[j]) {**

**i--;**

**j--;**

**}**

**if (j < 0) {**

**occurrences.push\_back(i + 1);**

**i += m + 1;**

**} else {**

**int skip = reverseFactor[j];**

**if (skip == 0) {**

**i += m;**

**} else {**

**i += std::max(skip, m - j);**

**}**

**}**

**}**

**return occurrences;**

**}**

**int main() {**

**std::string text = "ABABABA";**

**std::string pattern = "ABA";**

**std::vector<int> occurrences = turboReverseFactor(text, pattern);**

**if (occurrences.empty()) {**

**std::cout << "Pattern not found in text." << std::endl;**

**} else {**

**std::cout << "Pattern found at positions: ";**

**for (int pos : occurrences) {**

**std::cout << pos << " ";**

**}**

**std::cout << std::endl;**

**}**

**return 0;**

**}**

## Thuật toán Knuth-Morris-Pratt

* Trình bày về thuật toán

Thuật toán Knuth-Morris-Pratt (KMP) là một thuật toán tìm kiếm mẫu (pattern matching) trong một văn bản. Thuật toán này được sử dụng để tìm tất cả các vị trí xuất hiện của mẫu trong văn bản một cách hiệu quả.

Ý tưởng chính của thuật toán KMP:

* Tạo bảng lớn nhất khớp hệ tiền tố và hậu tố (longest proper prefix-suffix table) cho mẫu.
* Duyệt qua văn bản và mẫu để tìm các vị trí xuất hiện của mẫu:
* So sánh các ký tự của văn bản và mẫu.
* Nếu khớp, di chuyển tới ký tự tiếp theo trong văn bản và mẫu.

Nếu không khớp, sử dụng bảng lớn nhất khớp hệ tiền tố và hậu tố để xác định vị trí mới trong mẫu để tiếp tục so sánh.

* Đánh giá độ phức tạp của thuật toán
  + Độ phức tạp thời gian của thuật toán Knuth-Morris-Pratt là O(n + m), trong đó n là độ dài của văn bản và m là độ dài của mẫu.
  + Độ phức tạp không gian của thuật toán là O(m), do cần lưu trữ bảng lớn nhất khớp hệ tiền tố và hậu tố.
* Kiểm nghiệm thuật toán

Testcase 1:

Text: "ABCABCDABABCDABCD"

Pattern: "ABCD"

Kết quả mong đợi: Tìm thấy mẫu "ABCD" tại vị trí 3, 10, 14 trong văn bản.

Testcase 2:

Text: "AABAACAADAABAABA"

Pattern: "AABA"

Kết quả mong đợi: Tìm thấy mẫu "AABA" tại vị trí 0, 9, 12 trong văn bản.

Testcase 3:

Text: "Hello World"

Pattern: "Foo"

Kết quả mong đợi: Không tìm thấy mẫu "Foo" trong văn bản.

* Lập trình với thuật toán

**#include <iostream>**

**#include <string>**

**#include <set>**

**using namespace std;**

**set<string> findRepeatedSubstrings(const string& input\_string) {**

**set<string> repeated\_substrings;**

**for (int length = 2; length <= input\_string.length(); length++) {**

**for (int i = 0; i <= input\_string.length() - length; i++) {**

**string substring = input\_string.substr(i, length);**

**if (input\_string.find(substring, i + 1) != string::npos) {**

**repeated\_substrings.insert(substring);**

**}**

**}**

**}**

**return repeated\_substrings;**

**}**

**int main() {**

**string input\_str = "ababcababcab";**

**set<string> result = findRepeatedSubstrings(input\_str);**

**cout << "Repeated substrings:" << endl;**

**set<string>::iterator it;**

**for (it = result.begin(); it != result.end(); it++) {**

**cout << \*it << endl;**

**}**

**return 0;**

**}**

## Thuật toán Shift Or

* Trình bày về thuật toán

Thuật toán Shift Or được sử dụng để tìm kiếm các mẫu (pattern) trong văn bản (text). Thuật toán này dựa trên việc thực hiện các phép dịch và phép OR logic để so khớp các ký tự. Ý tưởng chính của thuật toán là sử dụng một mảng bit (bitmask) để lưu trữ các thông tin liên quan đến mẫu, giúp tăng tốc quá trình tìm kiếm.

Thuật toán Shift Or hoạt động theo các bước sau:

Tạo một mảng bitmask có kích thước bằng với bảng mã ASCII, ban đầu tất cả các bit đều được thiết lập thành 0.

Đối với mỗi ký tự trong mẫu, thiết lập bit tương ứng với ký tự đó trong bitmask thành 1.

Tìm kiếm mẫu trong văn bản bằng cách duyệt từ trái sang phải. Mỗi lần di chuyển sang phải, thực hiện phép dịch với bitmask và phép OR logic để so khớp ký tự hiện tại trong văn bản với ký tự tương ứng trong bitmask.

Nếu tất cả các bit trong bitmask đều bằng 0 tại một vị trí nào đó, đó chính là vị trí xuất hiện của mẫu trong văn bản.

* Đánh giá độ phức tạp của thuật toán
  + Độ phức tạp thời gian của thuật toán Shift Or là O(n + m), trong đó n là độ dài của văn bản và m là độ dài của mẫu.
  + Độ phức tạp không gian của thuật toán là O(1), vì thuật toán chỉ sử dụng một mảng bitmask cố định có kích thước cố định tương ứng với bảng mã ASCII.
* Kiểm nghiệm thuật toán

Text: "This is a test string."

Pattern: "test"

Kết quả mong đợi: Tìm thấy mẫu "test" tại vị trí 10 trong văn bản.

Text: "abababcababcabcabc"

Pattern: "abc"

Kết quả mong đợi: Tìm thấy mẫu "abc" tại các vị trí 5, 9, 11, 13, 15 trong văn bản.

Text: "Hello, world!"

Pattern: "xyz"

Kết quả mong đợi: Không tìm thấy mẫu "xyz" trong văn bản.

* Lập trình với thuật toán

**#include <iostream>**

**#include <vector>**

**#include <string>**

**std::vector<int> shiftOr(const std::string& text, const std::string& pattern) {**

**std::vector<int> matches;**

**int n = text.length();**

**int m = pattern.length();**

**std::vector<int> bitmask(256, 0);**

**// Thi?t l?p bitmask cho m?u**

**for (int i = 0; i < m; i++) {**

**bitmask[pattern[i]] |= (1 << i);**

**}**

**int state = 0;**

**for (int i = 0; i < n; i++) {**

**state = ((state << 1) | 1) & bitmask[text[i]];**

**if ((state & (1 << (m - 1))) != 0) {**

**matches.push\_back(i - m + 1);**

**}**

**}**

**return matches;**

**}**

**int main() {**

**std::string text = "abababcababcabcabc";**

**std::string pattern = "abc";**

**std::vector<int> matches = shiftOr(text, pattern);**

**std::cout << "Pattern found at positions: ";**

**for (int i = 0; i < matches.size(); i++) {**

**std::cout << matches[i] << " ";**

**}**

**std::cout << std::endl;**

**return 0;**

**}**

## Thuật toán Simon

* Trình bày về thuật toán

Thuật toán Simon là một thuật toán phân loại chuỗi đầu vào dựa trên việc tìm kiếm các mẫu chuỗi lặp lại trong đó. Thuật toán này được đặt tên theo tên Simon Singh, một nhà toán học và nhà văn người Anh.

Bước 1: Xác định các đoạn con độ dài từ 2 trở lên của chuỗi đầu vào và tạo ra một tập hợp con gọi là "đoạn con lặp lại".

Bước 2: Sắp xếp tập hợp đoạn con lặp lại theo thứ tự từ điển.

Bước 3: Tạo ra một tập hợp mới gọi là "đoạn con tương đồng" bằng cách so sánh từng cặp đoạn con lặp lại có chung đoạn con đầu tiên và đoạn con cuối cùng.

Bước 4: Sắp xếp tập hợp đoạn con tương đồng theo thứ tự từ điển.

Bước 5: Xác định đoạn con lặp lại dài nhất từ tập hợp đoạn con tương đồng.

Kết quả của thuật toán Simon là một chuỗi con lặp lại dài nhất có thể tìm thấy trong chuỗi đầu vào.

* Đánh giá độ phức tạp của thuật toán:

Độ phức tạp thời gian: Thuật toán Simon có độ phức tạp thời gian trung bình là O(n log n), trong đó n là độ dài của chuỗi đầu vào. Việc xác định các đoạn con đầu tiên và đoạn con cuối cùng trong chuỗi đầu vào mất O(n) thời gian. Quá trình sắp xếp và so sánh các đoạn con lặp lại và đoạn con tương đồng mất O(n log n) thời gian. Do đó, tổng độ phức tạp thời gian của thuật toán Simon là O(n log n).

Độ phức tạp không gian: Thuật toán Simon yêu cầu O(n) không gian để lưu trữ tập hợp các đoạn con lặp lại và đoạn con tương đồng trong quá trình thực hiện thuật toán.

* Kiểm nghiệm thuật toán:

Test Case 1: Chuỗi đầu vào không có đoạn con lặp lại:

Input: "abcdefg"

Expected Output: ""

Vì không có đoạn con lặp lại nào trong chuỗi đầu vào, kết quả trả về là chuỗi rỗng.

Test Case 2: Chuỗi đầu vào có một đoạn con lặp lại:

Input: "abcabcxyz"

Expected Output: "abc"

Giải thích: Đoạn con "abc" lặp lại hai lần trong chuỗi đầu vào. Đây là đoạn con lặp lại dài nhất và sẽ được trả về.

Test Case 3: Chuỗi đầu vào có nhiều đoạn con lặp lại:

Input: "ababcababcab"

Expected Output: "ababc"

Giải thích: Đoạn con "ababc" lặp lại ba lần trong chuỗi đầu vào. Đây là đoạn con lặp lại dài nhất và sẽ được trả về.

Test Case 4: Chuỗi đầu vào chỉ chứa một ký tự:

Input: "a"

Expected Output: ""

Giải thích: Vì không có đoạn con lặp lại nào trong chuỗi đầu vào, kết quả trả về là chuỗi rỗng.

Test Case 5: Chuỗi đầu vào chứa đoạn con lặp lại nhưng không liên tiếp:

Input: "ababcxyababcz"

Expected Output: "ababc"

* Lập trình theo thuật toán

**#include <iostream>**

**#include <string>**

**#include <set>**

**using namespace std;**

**set<string> findRepeatedSubstrings(const string& input\_string) {**

**set<string> repeated\_substrings;**

**for (int length = 2; length <= input\_string.length(); length++) {**

**for (int i = 0; i <= input\_string.length() - length; i++) {**

**string substring = input\_string.substr(i, length);**

**if (input\_string.find(substring, i + 1) != string::npos) {**

**repeated\_substrings.insert(substring);**

**}**

**}**

**}**

**return repeated\_substrings;**

**}**

**int main() {**

**string input\_str = "ababcababcab";**

**set<string> result = findRepeatedSubstrings(input\_str);**

**cout << "Repeated substrings:" << endl;**

**set<string>::iterator it;**

**for (it = result.begin(); it != result.end(); it++) {**

**cout << \*it << endl;**

**}**

**return 0;**

**}**

## Thuật toán Apostolico-Crochemore

* Trình bày về thuật toán
  + Thuật toán Apostolico-Crochemore kết hợp hai thuật toán khác là thuật toán Boyer-Moore và thuật toán Knuth-Morris-Pratt để thực hiện tìm kiếm mẫu.
  + Thuật toán này tận dụng thông tin từ cả hai thuật toán để tối ưu quá trình tìm kiếm.
  + Đầu tiên, thuật toán tìm kiếm từ phải sang trái bằng thuật toán Boyer-Moore để loại bỏ các vị trí không cần thiết trong quá trình tìm kiếm.
  + Sau đó, thuật toán chuyển sang thuật toán Knuth-Morris-Pratt để tiếp tục tìm kiếm mẫu từ trái sang phải.
* Đánh giá độ phức tạp của thuật toán
  + Độ phức tạp thời gian của thuật toán Apostolico-Crochemore là O(n + m), trong đó n là độ dài của chuỗi nhập và m là độ dài của mẫu.
  + Độ phức tạp không gian của thuật toán là O(m), với m là độ dài của mẫu.
* Kiểm nghiệm thuật toán
  + Testcase 1: Chuỗi và mẫu đều trống ("", ""), kết quả mong đợi là tìm thấy mẫu tại vị trí 0.
  + Testcase 2: Chuỗi trống và mẫu không trống ("", "abc"), kết quả mong đợi là không tìm thấy mẫu.
  + Testcase 3: Chuỗi không trống và mẫu trống ("abcdef", ""), kết quả mong đợi là tìm thấy mẫu tại vị trí 0.
  + Testcase 4: Chuỗi và mẫu khác nhau, không tìm thấy mẫu ("abcdef", "xyz"), kết quả mong đợi là không tìm thấy mẫu.
  + Testcase 5: Chuỗi và mẫu khớp hoàn toàn ("abcdef", "abc"), kết quả mong đợi là tìm thấy mẫu tại vị trí 0.
* Lập trình với thuật toán

**#include <iostream>**

**#include <vector>**

**#include <string>**

**std::vector<int> computeLPS(const std::string& pattern) {**

**int m = pattern.length();**

**std::vector<int> lps(m);**

**int len = 0;**

**int i = 1;**

**while (i < m) {**

**if (pattern[i] == pattern[len]) {**

**len++;**

**lps[i] = len;**

**i++;**

**} else {**

**if (len != 0) {**

**len = lps[len - 1];**

**} else {**

**lps[i] = 0;**

**i++;**

**}**

**}**

**}**

**return lps;**

**}**

**std::vector<int> apostolicoCrochemore(const std::string& text, const std::string& pattern) {**

**std::vector<int> matches;**

**int n = text.length();**

**int m = pattern.length();**

**std::vector<int> lps = computeLPS(pattern);**

**int i = 0;**

**int j = 0;**

**while (i < n) {**

**if (pattern[j] == text[i]) {**

**i++;**

**j++;**

**if (j == m) {**

**matches.push\_back(i - j);**

**j = lps[j - 1];**

**}**

**} else {**

**if (j != 0) {**

**j = lps[j - 1];**

**} else {**

**i++;**

**}**

**}**

**}**

**return matches;**

**}**

**int main() {**

**std::string text = "ABCABCDABABCDABCD";**

**std::string pattern = "ABCD";**

**std::vector<int> matches = apostolicoCrochemore(text, pattern);**

**std::cout << "Pattern found at positions: ";**

**for (int i = 0; i < matches.size(); i++) {**

**std::cout << matches[i] << " ";**

**}**

**std::cout << std::endl;**

**return 0;**

**}**

## Thuật toán Karp-Rabin

* Trình bày về thuật toán

Thuật toán Karp-Rabin là một thuật toán tìm kiếm mẫu (pattern matching) trong một văn bản. Thuật toán này sử dụng phép toán modulo để tính toán giá trị băm (hash value) của các phần tử trong văn bản và mẫu.

Ý tưởng chính của thuật toán Karp-Rabin:

Tính toán giá trị băm cho mẫu.

Duyệt qua văn bản và tính toán giá trị băm cho các khối ký tự có cùng độ dài với mẫu.

So sánh giá trị băm của mẫu với giá trị băm của các khối ký tự trong văn bản. Nếu giá trị băm khớp, thực hiện kiểm tra thủ công để xác định xem có sự khớp thực sự hay không.

* Đánh giá độ phức tạp của thuật toán
  + Độ phức tạp thời gian trung bình của thuật toán Karp-Rabin là O(n + m), trong đó n là độ dài của văn bản và m là độ dài của mẫu.
  + Độ phức tạp không gian của thuật toán là O(1).
* Kiểm nghiệm thuật toán

Testcase 1:

Text: "ABCABCDABABCDABCD"

Pattern: "ABCD"

Kết quả mong đợi: Tìm thấy mẫu "ABCD" tại vị trí 3, 10, 14 trong văn bản.

Testcase 2:

Text: "AABAACAADAABAABA"

Pattern: "AABA"

Kết quả mong đợi: Tìm thấy mẫu "AABA" tại vị trí 0, 9, 12 trong văn bản.

Testcase 3:

Text: "Hello World"

Pattern: "Foo"

Kết quả mong đợi: Không tìm thấy mẫu "Foo" trong văn bản.

* Lập trình với thuật toán

**#include <iostream>**

**#include <string>**

**#include <cmath>**

**const int PRIME = 101; // S? nguyên t? dùng cho phép toán modulo**

**bool areEqual(const std::string& text, const std::string& pattern, int start) {**

**int m = pattern.length();**

**for (int i = 0; i < m; i++) {**

**if (text[start + i] != pattern[i]) {**

**return false;**

**}**

**}**

**return true;**

**}**

**void karpRabin(const std::string& text, const std::string& pattern) {**

**int n = text.length();**

**int m = pattern.length();**

**int patternHash = 0; // Giá tr? bam c?a m?u**

**int textHash = 0; // Giá tr? bam c?a do?n van b?n hi?n t?i**

**// Tính toán giá tr? bam cho m?u và do?n van b?n d?u tiên có d? dài m**

**for (int i = 0; i < m; i++) {**

**patternHash += pattern[i] \* std::pow(PRIME, i);**

**textHash += text[i] \* std::pow(PRIME, i);**

**}**

**for (int i = 0; i <= n - m; i++) {**

**if (patternHash == textHash && areEqual(text, pattern, i)) {**

**std::cout << "Pattern found at position: " << i << std::endl;**

**}**

**// Tính toán giá tr? bam c?a do?n van b?n ti?p theo**

**if (i < n - m) {**

**textHash = (textHash - text[i]) / PRIME + text[i + m] \* std::pow(PRIME, m - 1);**

**}**

**}**

**}**

**int main() {**

**std::string text = "ABCABCDABABCDABCD";**

**std::string pattern = "ABCD";**

**karpRabin(text, pattern);**

**return 0;**

**}**

# **Tìm kiếm mẫu từ phải sang trái**

## Thuật toán Turbo BM

Thuật toán Turbo BM là một thuật toán tìm kiếm chuỗi con hiệu quả dựa trên việc kết hợp hai thuật toán nổi tiếng là Boyer-Moore và Apostolico-Giancarlo. Nó cải tiến thuật toán Boyer-Moore bằng cách sử dụng các bảng phụ và các kỹ thuật tối ưu khác để tăng tốc độ tìm kiếm.

Thuật toán Turbo BM sử dụng hai bảng phụ là bảng chuyển đổi dự đoán và bảng chuyển đổi tối ưu. Bảng chuyển đổi dự đoán được sử dụng để đưa ra các dự đoán về vị trí tiếp theo của chuỗi con trong chuỗi đầu vào. Bảng chuyển đổi tối ưu được sử dụng để di chuyển qua các vị trí có khả năng cao chứa chuỗi con. Bằng cách kết hợp cả hai bảng, thuật toán giảm thiểu số lượng so sánh cần thiết để tìm kiếm chuỗi con.

Thuật toán Turbo BM hoạt động bằng cách so sánh các ký tự từ phải sang trái của chuỗi con với chuỗi đầu vào và di chuyển theo các quy tắc của thuật toán Boyer-Moore và Apostolico-Giancarlo. Khi tìm thấy một khớp hoàn hảo, thuật toán trả về vị trí của chuỗi con trong chuỗi đầu vào hoặc một danh sách các vị trí nếu có nhiều kết quả.

Đánh giá độ phức tạp của thuật toán:

* Độ phức tạp trong trường hợp tốt nhất: O(m/n), trong đó m là độ dài của chuỗi con cần tìm và n là độ dài của chuỗi đầu vào.
* Độ phức tạp trong trường hợp trung bình và xấu nhất: O(m + n), trong đó m là độ dài của chuỗi con cần tìm và n là độ dài của chuỗi đầu vào.

Kiểm nghiệm thuật toán Turbo BM:

Chuỗi đầu vào: "ababcababcabc"

Chuỗi con: "abc"

Kết quả kỳ vọng: Tìm thấy chuỗi con tại vị trí 2.

Chuỗi đầu vào: "abababababab"

Chuỗi con: "aba"

Kết quả kỳ vọng: Tìm thấy chuỗi con tại các vị trí 0, 2, 4, 6, 8, 10.

Chuỗi đầu vào: "aaaaaa"

Chuỗi con: "b"

Kết quả kỳ vọng: Không tìm thấy chuỗi con trong chuỗi đầu vào.

Chuỗi đầu vào: "abcdef"

Chuỗi con: "defg"

Kết quả kỳ vọng: Không tìm thấy chuỗi con trong chuỗi đầu vào.

**#include <iostream>**

**#include <string>**

**#include <vector>**

**using namespace std;**

**vector<int> turboBM(const string& text, const string& pattern) {**

**vector<int> positions;**

**int n = text.length();**

**int m = pattern.length();**

**// Tạo bảng chuyển đổi dự đoán**

**vector<int> predictShift(256, m);**

**for (int i = 0; i < m - 1; i++) {**

**predictShift[pattern[i]] = m - i - 1;**

**}**

**// Tạo bảng chuyển đổi tối ưu**

**vector<int> optimalShift(m, 0);**

**int lastPrefixPosition = m;**

**for (int i = m - 1; i >= 0; i--) {**

**if (isPrefix(pattern, i + 1)) {**

**lastPrefixPosition = i + 1;**

**}**

**optimalShift[i] = lastPrefixPosition + m - i - 1;**

**}**

**for (int i = 0; i < m - 1; i++) {**

**int suffixLength = longestSuffix(pattern, i);**

**if (pattern[i - suffixLength] != pattern[m - 1 - suffixLength]) {**

**optimalShift[m - 1 - suffixLength] = m - 1 - i + suffixLength;**

**}**

**}**

**// Tiến hành tìm kiếm**

**int i = m - 1;**

**while (i < n) {**

**int j = m - 1;**

**while (j >= 0 && text[i] == pattern[j]) {**

**i--;**

**j--;**

**}**

**if (j < 0) {**

**positions.push\_back(i + 1);**

**i += m + 1;**

**} else {**

**i += max(predictShift[text[i]], optimalShift[j]);**

**}**

**}**

**return positions;**

**}**

**bool isPrefix(const string& pattern, int position) {**

**int patternLength = pattern.length();**

**for (int i = position, j = 0; i < patternLength; i++, j++) {**

**if (pattern[i] != pattern[j]) {**

**return false;**

**}**

**}**

**return true;**

**}**

**int longestSuffix(const string& pattern, int position) {**

**int length = 0;**

**int i = position;**

**int j = pattern.length() - 1;**

**while (i >= 0 && pattern[i] == pattern[j]) {**

**length++;**

**i--;**

**j--;**

**}**

**return length;**

**}**

**int main() {**

**string text = "ababcababcabc";**

**string pattern = "abc";**

**vector<int> positions = turboBM(text, pattern);**

**if (positions.empty()) {**

**cout << "Pattern not found in the text." << endl;**

**} else {**

**cout << "Pattern found at positions:";**

**for (int pos : positions) {**

**cout << " " << pos;**

**}**

**cout << endl;**

**}**

**return 0;**

**}**

## Thuật toán Zhu-Takaoka

Thuật toán Zhu-Takaoka là một thuật toán tìm kiếm chuỗi con hiệu quả được sử dụng để tìm kiếm tất cả các xuất hiện của một chuỗi con trong một chuỗi đầu vào. Thuật toán này kết hợp giữa kỹ thuật tìm kiếm dựa trên hàm băm và tìm kiếm dựa trên tổ hợp của các chuỗi con.

Thuật toán Zhu-Takaoka hoạt động bằng cách xác định các chỉ mục tiềm năng trong chuỗi đầu vào sử dụng hàm băm, sau đó kiểm tra các tổ hợp của các chuỗi con để tìm kiếm các kết quả chính xác. Quá trình này được lặp lại cho tất cả các chỉ mục tiềm năng, đảm bảo rằng tất cả các xuất hiện của chuỗi con được tìm thấy.

Đánh giá độ phức tạp của thuật toán:

* Độ phức tạp trong trường hợp tốt nhất: O(n)
* Độ phức tạp trong trường hợp trung bình: O(n + m)
* Độ phức tạp trong trường hợp xấu nhất: O(n + m), trong đó n là độ dài của chuỗi đầu vào và m là độ dài của chuỗi con cần tìm.

Kiểm nghiệm thuật toán Zhu-Takaoka:

Chuỗi đầu vào: "ababcababcabc"

Chuỗi con: "abc"

Kết quả kỳ vọng: Tìm thấy chuỗi con tại các vị trí 2, 5 và 8.

Chuỗi đầu vào: "abababababab"

Chuỗi con: "aba"

Kết quả kỳ vọng: Tìm thấy chuỗi con tại các vị trí 1, 3, 5, 7 và 9.

Chuỗi đầu vào: "abcdefg"

Chuỗi con: "xyz"

Kết quả kỳ vọng: Không tìm thấy chuỗi con trong chuỗi đầu vào.

**#include <iostream>**

**#include <string>**

**#include <unordered\_set>**

**using namespace std;**

**void zhuTakaokaSearch(const string& text, const string& pattern) {**

**int n = text.length();**

**int m = pattern.length();**

**unordered\_set<int> result;**

**for (int i = 0; i <= n - m; i++) {**

**if (text.substr(i, m) == pattern) {**

**result.insert(i);**

**}**

**}**

**if (!result.empty()) {**

**cout << "Pattern found at positions: ";**

**for (int pos : result) {**

**cout << pos << " ";**

**}**

**cout << endl;**

**} else {**

**cout << "Pattern not found in the text." << endl;**

**}**

**}**

**int main() {**

**string text = "ababcababcabc";**

**string pattern = "abc";**

**zhuTakaokaSearch(text, pattern);**

**return 0;**

**}**

## Thuật toán Berry-Ravindran

Thuật toán Berry-Ravindran là một thuật toán được sử dụng để tìm kiếm chuỗi con trong chuỗi. Nó sử dụng phương pháp kết hợp giữa việc xác định các vị trí tiềm năng của chuỗi con và việc tìm kiếm chi tiết để xác định vị trí chính xác của chuỗi con trong chuỗi.

Thuật toán Berry-Ravindran bao gồm hai giai đoạn chính: giai đoạn tiền xử lý và giai đoạn tìm kiếm. Giai đoạn tiền xử lý nhằm xây dựng một bảng phụ để xác định các vị trí tiềm năng của chuỗi con. Giai đoạn tìm kiếm sử dụng bảng phụ này để tìm kiếm chi tiết chuỗi con trong chuỗi.

Đánh giá độ phức tạp của thuật toán:

* Độ phức tạp trong trường hợp tốt nhất: O(m)
* Độ phức tạp trong trường hợp trung bình: O(n)
* Độ phức tạp trong trường hợp xấu nhất: O(nm), trong đó n là độ dài của chuỗi đầu vào và m là độ dài của chuỗi con cần tìm.

Kiểm nghiệm thuật toán Berry-Ravindran:

Chuỗi đầu vào: "ababcababcabc"

Chuỗi con: "abc"

Kết quả kỳ vọng: Tìm thấy chuỗi con tại các vị trí 2, 5 và 8.

Chuỗi đầu vào: "abababababab"

Chuỗi con: "aba"

Kết quả kỳ vọng: Tìm thấy chuỗi con tại các vị trí 1, 3, 5, 7 và 9.

Chuỗi đầu vào: "abcdefg"

Chuỗi con: "xyz"

Kết quả kỳ vọng: Không tìm thấy chuỗi con trong chuỗi đầu vào.

**#include <iostream>**

**#include <string>**

**using namespace std;**

**void berryRavindranSearch(const string& text, const string& pattern) {**

**int n = text.length();**

**int m = pattern.length();**

**int i = 0;**

**while (i <= n - m) {**

**int j = m - 1;**

**while (j >= 0 && pattern[j] == text[i + j]) {**

**j--;**

**}**

**if (j < 0) {**

**cout << "Pattern found at position " << i << endl;**

**i++;**

**} else {**

**i += max(1, j - pattern.find\_last\_of(text[i + j]));**

**}**

**}**

**}**

**int main() {**

**string text = "ababcababcabc";**

**string pattern = "abc";**

**berryRavindranSearch(text, pattern);**

**return 0;**

**}**

## Thuật toán Quick Search

Thuật toán Quick Search là một thuật toán tìm kiếm mẫu trong chuỗi (pattern matching) được sử dụng để tìm kiếm một mẫu (pattern) trong một chuỗi văn bản (text). Thuật toán này kết hợp sử dụng một bảng dịch (shift table) và một bảng tần suất (frequency table) để tăng tốc quá trình tìm kiếm.

Sơ lược về thuật toán Quick Search:

* Tiền xử lý: Xây dựng bảng dịch, một bảng chứa thông tin về sự dịch chuyển của mẫu khi không khớp với văn bản. Bảng dịch dựa trên tần suất của các ký tự trong mẫu.
* Tìm kiếm: Bắt đầu tại vị trí bắt đầu của mẫu trong văn bản và di chuyển một khoảng dựa trên bảng dịch và bảng tần suất để tìm kiếm mẫu.
* So sánh: So sánh các ký tự từ phải sang trái của mẫu với văn bản.
* Nếu các ký tự khớp, tiếp tục so sánh với các ký tự tiếp theo theo chiều từ phải sang trái.
* Nếu không khớp, sử dụng thông tin từ bảng dịch để di chuyển tới vị trí tiếp theo có thể chứa một phần của mẫu.
* Lặp lại quá trình cho đến khi tìm thấy mẫu hoặc hết văn bản.

Đánh giá độ phức tạp của thuật toán:

* Độ phức tạp trong trường hợp tốt nhất là O(n/m), trong đó n là độ dài của chuỗi văn bản và m là độ dài của mẫu. Trong trường hợp này, thuật toán chỉ cần một lần so sánh tại mỗi vị trí.
* Độ phức tạp trong trường hợp xấu nhất là O(n \* m), với n là độ dài của chuỗi văn bản và m là độ dài của mẫu. Trong trường hợp xấu nhất, tất cả các ký tự trong văn bản đều khớp với ký tự cuối cùng của mẫu, khiến thuật toán phải so sánh tại mỗi vị trí.

Kiểm nghiệm thuật toán Quick Search:

Text: "ABABABA"

Pattern: "ABA"

Kết quả: Tìm thấy mẫu tại các vị trí 0, 2, 4.

Text: "ABCDEF"

Pattern: "XYZ"

Kết quả: Không tìm thấy mẫu trong văn bản.

Text: "AAAAA"

Pattern: "AA"

Kết quả: Tìm thấy mẫu tại các vị trí 0, 1, 2, 3.

Lập trình theo thuật toán:

**#include <iostream>**

**#include <string>**

**#include <vector>**

**std::vector<int> quickSearch(const std::string& text, const std::string& pattern) {**

**std::vector<int> occurrences;**

**int n = text.length();**

**int m = pattern.length();**

**// Xây dựng bảng dịch**

**std::vector<int> shiftTable(256, m);**

**for (int i = 0; i < m - 1; i++) {**

**shiftTable[pattern[i]] = m - i - 1;**

**}**

**int i = m - 1;**

**while (i < n) {**

**int j = m - 1;**

**while (j >= 0 && text[i] == pattern[j]) {**

**i--;**

**j--;**

**}**

**if (j < 0) {**

**occurrences.push\_back(i + 1);**

**i += m;**

**} else {**

**i += shiftTable[text[i]];**

**}**

**}**

**return occurrences;**

**}**

**int main() {**

**std::string text = "ABABABA";**

**std::string pattern = "ABA";**

**std::vector<int> occurrences = quickSearch(text, pattern);**

**if (occurrences.empty()) {**

**std::cout << "Pattern not found in text." << std::endl;**

**} else {**

**std::cout << "Pattern found at positions: ";**

**for (int pos : occurrences) {**

**std::cout << pos << " ";**

**}**

**std::cout << std::endl;**

**}**

**return 0;**

**}**

## Thuật toán Reverse Colussi

* Trình bày về thuật toán
  + Reverse Colussi là một biến thể của thuật toán Colussi, và nó sử dụng một bảng chuyển đổi để tối ưu quá trình tìm kiếm.
  + Thuật toán Reverse Colussi hoạt động bằng cách so sánh các ký tự từ phải sang trái, bắt đầu từ cuối mẫu và cuối chuỗi.
  + Khi có sự không khớp xảy ra, thuật toán sử dụng bảng chuyển đổi để xác định bước nhảy tối ưu để di chuyển.
* Đánh giá độ phức tạp của thuật toán
  + Độ phức tạp thời gian của thuật toán Reverse Colussi trong trường hợp xấu nhất là O(mn), trong đó m là độ dài của mẫu và n là độ dài của chuỗi nhập.
  + Độ phức tạp không gian của thuật toán là O(m), với m là độ dài của mẫu.
* Kiểm nghiệm thuật toán
  + Testcase 1: Chuỗi và mẫu đều trống ("", ""), kết quả mong đợi là tìm thấy mẫu tại vị trí 0.
  + Testcase 2: Chuỗi trống và mẫu không trống ("", "abc"), kết quả mong đợi là không tìm thấy mẫu.
  + Testcase 3: Chuỗi không trống và mẫu trống ("abcdef", ""), kết quả mong đợi là tìm thấy mẫu tại vị trí 0.
  + Testcase 4: Chuỗi và mẫu khác nhau, không tìm thấy mẫu ("abcdef", "xyz"), kết quả mong đợi là không tìm thấy mẫu.
  + Testcase 5: Chuỗi và mẫu khớp hoàn toàn ("abcdef", "abc"), kết quả mong đợi là tìm thấy mẫu tại vị trí 0.
* Lập trình với thuật toán

**#include <iostream>**

**#include <vector>**

**#include <string>**

**std::vector<int> build\_table(const std::string& pattern) {**

**int m = pattern.length();**

**std::vector<int> table(m + 1, m);**

**for (int i = 0; i < m; i++) {**

**table[m - i - 1] = m - i;**

**for (int j = m - i; j < m; j++) {**

**if (pattern[m - i - 1] == pattern[j]) {**

**table[m - i - 1] = j - (m - i - 1);**

**break;**

**}**

**}**

**}**

**return table;**

**}**

**std::vector<int> reverse\_colussi(const std::string& text, const std::string& pattern) {**

**std::vector<int> matches;**

**int n = text.length();**

**int m = pattern.length();**

**std::vector<int> table = build\_table(pattern);**

**int i = m - 1;**

**int j = i;**

**int k = i;**

**while (j >= 0 && i < n) {**

**if (text[i] == pattern[j]) {**

**i--;**

**j--;**

**} else {**

**i += table[j];**

**j = m - 1;**

**}**

**}**

**if (j < 0) {**

**matches.push\_back(i + 1);**

**k = m - 1;**

**}**

**while (i < n) {**

**if (text[i] == pattern[k]) {**

**i++;**

**k--;**

**} else {**

**i += table[m - k - 1];**

**k = m - 1;**

**}**

**if (k < 0) {**

**matches.push\_back(i - m);**

**k = m - 1;**

**}**

**}**

**return matches;**

**}**

**int main() {**

**std::string text = "ABCABCDABABCDABCD";**

**std::string pattern = "ABCD";**

**std::vector<int> matches = reverse\_colussi(text, pattern);**

**std::cout << "Pattern found at positions: ";**

**for (int i = 0; i < matches.size(); i++) {**

**std::cout << matches[i] << " ";**

**}**

**std::cout << std::endl;**

**return 0;**

**}**

## Thuật toán Tuned Boyer-Moore

* Trình bày về thuật toán
  + Tuned Boyer-Moore kết hợp hai phương pháp tối ưu của Boyer-Moore: Bad Character Shift (di chuyển dựa trên ký tự không phù hợp) và Good Suffix Shift (di chuyển dựa trên hậu tố tốt).
  + Thuật toán sử dụng bảng Bad Character và một mảng Good Suffix để xác định các vị trí tiềm năng để tìm kiếm.
  + Thay vì di chuyển theo từng ký tự, thuật toán thực hiện việc di chuyển lớn hơn một bước bằng cách sử dụng thông tin từ cả Bad Character và Good Suffix.
* Đánh giá độ phức tạp của thuật toán
  + Độ phức tạp thời gian của thuật toán Tuned Boyer-Moore trong trường hợp xấu nhất là O(m + n), trong đó m là độ dài của mẫu cần tìm kiếm và n là độ dài của chuỗi nhập.
  + Độ phức tạp không gian của thuật toán là O(m), với m là độ dài của mẫu.
* Kiểm nghiệm thuật toán
  + Testcase 1: Chuỗi và mẫu đều trống ("", ""), kết quả mong đợi là tìm thấy mẫu tại vị trí 0.
  + Testcase 2: Chuỗi trống và mẫu không trống ("", "abc"), kết quả mong đợi là không tìm thấy mẫu.
  + Testcase 3: Chuỗi không trống và mẫu trống ("abcdef", ""), kết quả mong đợi là tìm thấy mẫu tại vị trí 0.
  + Testcase 4: Chuỗi và mẫu khác nhau, không tìm thấy mẫu ("abcdef", "xyz"), kết quả mong đợi là không tìm thấy mẫu.
  + Testcase 5: Chuỗi và mẫu khớp hoàn toàn ("abcdef", "abc"), kết quả mong đợi là tìm thấy mẫu tại vị trí 0.
* Lập trình với thuật toán

**#include <iostream>**

**#include <vector>**

**#include <string>**

**std::vector<int> build\_table(const std::string& pattern) {**

**int m = pattern.length();**

**std::vector<int> table(m + 1, m);**

**for (int i = 0; i < m; i++) {**

**table[m - i - 1] = m - i;**

**for (int j = m - i; j < m; j++) {**

**if (pattern[m - i - 1] == pattern[j]) {**

**table[m - i - 1] = j - (m - i - 1);**

**break;**

**}**

**}**

**}**

**return table;**

**}**

**std::vector<int> reverse\_colussi(const std::string& text, const std::string& pattern) {**

**std::vector<int> matches;**

**int n = text.length();**

**int m = pattern.length();**

**std::vector<int> table = build\_table(pattern);**

**int i = m - 1;**

**int j = i;**

**int k = i;**

**while (j >= 0 && i < n) {**

**if (text[i] == pattern[j]) {**

**i--;**

**j--;**

**} else {**

**i += table[j];**

**j = m - 1;**

**}**

**}**

**if (j < 0) {**

**matches.push\_back(i + 1);**

**k = m - 1;**

**}**

**while (i < n) {**

**if (text[i] == pattern[k]) {**

**i++;**

**k--;**

**} else {**

**i += table[m - k - 1];**

**k = m - 1;**

**}**

**if (k < 0) {**

**matches.push\_back(i - m);**

**k = m - 1;**

**}**

**}**

**return matches;**

**}**

**int main() {**

**std::string text = "ABCABCDABABCDABCD";**

**std::string pattern = "ABCD";**

**std::vector<int> matches = reverse\_colussi(text, pattern);**

**std::cout << "Pattern found at positions: ";**

**for (int i = 0; i < matches.size(); i++) {**

**std::cout << matches[i] << " ";**

**}**

**std::cout << std::endl;**

**return 0;**

**}**

## Thuật toán Reverse Factor algorithm

* Trình bày về thuật toán
  + Thuật toán "Reverse Factor" được sử dụng để đảo ngược thứ tự các yếu tố trong một danh sách.
  + Đầu vào của thuật toán là một danh sách các yếu tố.
  + Thuật toán sẽ hoán đổi vị trí giữa các yếu tố, sao cho yếu tố đầu tiên trở thành yếu tố cuối cùng, yếu tố thứ hai trở thành yếu tố thứ hai từ cuối cùng, và tiếp tục cho đến yếu tố cuối cùng trở thành yếu tố đầu tiên.
* Đánh giá độ phức tạp của thuật toán
  + Độ phức tạp thời gian của thuật toán "Reverse Factor" là O(n), trong đó n là số lượng yếu tố trong danh sách. Vì thuật toán chỉ duyệt qua danh sách một lần và thực hiện hoán đổi vị trí giữa các yếu tố.
  + Độ phức tạp không gian của thuật toán là O(1), vì không cần thêm không gian bộ nhớ phụ phức tạp.
* Kiểm nghiệm thuật toán
  + Testcase 1: danh sách trống ([]), kết quả mong đợi vẫn là danh sách trống ([]).
  + Testcase 2: danh sách gồm một yếu tố ([1]), kết quả mong đợi vẫn là danh sách gồm một yếu tố ([1]).
  + Testcase 3: danh sách gồm nhiều yếu tố ([1, 2, 3, 4, 5]), kết quả mong đợi là danh sách đảo ngược ([5, 4, 3, 2, 1]).
* Lập trình với thuật toán

**#include <iostream>**

**#include <vector>**

**std::vector<int> reverse\_factor(std::vector<int> lst) {**

**int start = 0;**

**int end = lst.size() - 1;**

**while (start < end) {**

**std::swap(lst[start], lst[end]);**

**start++;**

**end--;**

**}**

**return lst;**

**int main() {**

**std::vector<int> input\_list = {1, 2, 3, 4, 5};**

**std::vector<int> reversed\_list = reverse\_factor(input\_list);**

**// In danh sách đã đảo ngược**

**for (int num : reversed\_list) {**

**std::cout << num << " ";**

**}**

**std::cout << std::endl;**

**return 0;**

**}**

# **Tìm kiếm mẫu từ vị trí xác định**

## Thuật toán KMP Skip Search

Trình bày sơ lược về thuật toán: Thuật toán KMP Skip Search (Knuth-Morris-Pratt Skip Search) là một biến thể của thuật toán Knuth-Morris-Pratt (KMP) để tìm kiếm mẫu trong văn bản. Nó sử dụng một bảng chuyển đổi dựa trên khoảng cách của các ký tự không khớp để thực hiện nhảy qua các vị trí không cần thiết trong quá trình tìm kiếm. Điều này giúp cải thiện hiệu suất của thuật toán KMP truyền thống.

Thuật toán KMP Skip Search thực hiện các bước sau:

* Tạo bảng chuyển đổi tiền tố (prefix) của mẫu.
* Tạo bảng chuyển đổi khoảng cách (skip) dựa trên bảng chuyển đổi tiền tố.
* Tiến hành tìm kiếm mẫu trong văn bản bằng cách so sánh từng ký tự.
* Khi có ký tự không khớp, sử dụng bảng chuyển đổi skip để nhảy qua các vị trí không cần thiết.

Đánh giá độ phức tạp của thuật toán: Độ phức tạp thời gian của thuật toán KMP Skip Search là O(n + m), trong đó n là độ dài của văn bản và m là độ dài của mẫu. Độ phức tạp không gian là O(m), do phải lưu trữ bảng chuyển đổi tiền tố và bảng chuyển đổi skip.

Kiểm nghiệm thuật toán KMP Skip Search:

Văn bản: "ABCABDABACDABABCABDE"

Mẫu: "ABABCABDE"

Kết quả: Mẫu được tìm thấy tại vị trí 10 trong văn bản.

Văn bản: "ABCDABCDABDE"

Mẫu: "ABDE"

Kết quả: Mẫu được tìm thấy tại vị trí 9 trong văn bản.

Văn bản: "ABCDABCDABDE"

Mẫu: "XYZ"

Kết quả: Mẫu không được tìm thấy trong văn bản.

**#include <iostream>**

**#include <string>**

**#include <vector>**

**using namespace std;**

**vector<int> computePrefix(const string& pattern) {**

**int m = pattern.length();**

**vector<int> prefix(m, 0);**

**int j = 0;**

**for (int i = 1; i < m; i++) {**

**while (j > 0 && pattern[i] != pattern[j]) {**

**j = prefix[j - 1];**

**}**

**if (pattern[i] == pattern[j]) {**

**j++;**

**}**

**prefix[i] = j;**

**}**

**return prefix;**

**}**

**vector<int> kmpSkipSearch(const string& text, const string& pattern) {**

**vector<int> positions;**

**int n = text.length();**

**int m = pattern.length();**

**vector<int> prefix = computePrefix(pattern);**

**vector<int> skip(m, m);**

**for (int i = m - 1; i >= 1; i--) {**

**int j = prefix[i] - 1;**

**skip[j] = min(skip[j], i);**

**}**

**int i = 0;**

**while (i <= n - m) {**

**int j = 0;**

**while (j < m && text[i + j] == pattern[j]) {**

**j++;**

**}**

**if (j == m) {**

**positions.push\_back(i);**

**}**

**if (j == 0) {**

**i++;**

**} else {**

**i += skip[j - 1];**

**}**

**}**

**return positions;**

**}**

**int main() {**

**string text = "ABCABDABACDABABCABDE";**

**string pattern = "ABABCABDE";**

**vector<int> positions = kmpSkipSearch(text, pattern);**

**if (positions.empty()) {**

**cout << "Pattern not found in the text." << endl;**

**} else {**

**cout << "Pattern found at positions:";**

**for (int pos : positions) {**

**cout << " " << pos;**

**}**

**cout << endl;**

**}**

**return 0;**

**}**

## Thuật toán Alpha Skip Search

Trình bày sơ lược về thuật toán: Thuật toán Alpha Skip Search là một biến thể của thuật toán Skip List được áp dụng để tìm kiếm phần tử trong một danh sách liên kết. Skip List là một cấu trúc dữ liệu được sắp xếp và cho phép tìm kiếm có độ phức tạp trung bình là O(log n), tương tự như cây nhị phân cân bằng. Alpha Skip Search là một phiên bản tối ưu hóa của Skip List, sử dụng một bước nhảy được tính toán để giảm số lượng so sánh cần thực hiện trong quá trình tìm kiếm.

Thuật toán Alpha Skip Search thực hiện các bước sau:

* Xây dựng cây Skip List từ danh sách dữ liệu đầu vào.
* Tính toán bước nhảy alpha dựa trên kích thước của danh sách.
* Tiến hành tìm kiếm phần tử bằng cách thực hiện nhảy alpha và so sánh giá trị tại mỗi nút.
* Nếu phần tử được tìm thấy, trả về vị trí của nó. Nếu không, tiếp tục tìm kiếm theo hướng thích hợp.

Đánh giá độ phức tạp của thuật toán: Độ phức tạp thời gian trung bình của thuật toán Alpha Skip Search là O(log n), với n là kích thước của danh sách. Tuy nhiên, độ phức tạp trong trường hợp xấu nhất có thể là O(n), khi phần tử cần tìm kiếm nằm ở cuối danh sách và không có bước nhảy nào có thể được sử dụng.

Kiểm nghiệm thuật toán Alpha Skip Search:

Danh sách: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

Phần tử cần tìm: 9

Kết quả: Phần tử 9 được tìm thấy tại vị trí 4.

Danh sách: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

Phần tử cần tìm: 15

Kết quả: Phần tử 15 không được tìm thấy trong danh sách.

Danh sách: [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]

Phần tử cần tìm: 35

Kết quả: Phần tử 35 được tìm thấy tại vị trí 6.

**#include <iostream>**

**#include <vector>**

**#include <random>**

**using namespace std;**

**struct Node {**

**int value;**

**vector<Node\*> next;**

**Node(int val, int level) : value(val), next(level, nullptr) {}**

**};**

**class SkipList {**

**private:**

**Node\* head;**

**int maxLevel;**

**int level;**

**float probability;**

**random\_device rd;**

**mt19937 gen;**

**uniform\_real\_distribution<> dis;**

**public:**

**SkipList(int maxLvl, float p) : maxLevel(maxLvl), probability(p), level(1), gen(rd()), dis(0, 1) {**

**head = new Node(0, maxLevel);**

**}**

**int randomLevel() {**

**int lvl = 1;**

**while (dis(gen) < probability && lvl < maxLevel) {**

**lvl++;**

**}**

**return lvl;**

**}**

**void insert(int val) {**

**int newLevel = randomLevel();**

**if (newLevel > level) {**

**for (int i = level + 1; i <= newLevel; i++) {**

**head->next.push\_back(nullptr);**

**}**

**level = newLevel;**

**}**

**Node\* newNode = new Node(val, newLevel);**

**Node\* current = head;**

**for (int i = level - 1; i >= 0; i--) {**

**while (current->next[i] != nullptr && current->next[i]->value < val) {**

**current = current->next[i];**

**}**

**if (i < newLevel) {**

**newNode->next[i] = current->next[i];**

**current->next[i] = newNode;**

**}**

**}**

**}**

**bool search(int val) {**

**Node\* current = head;**

**for (int i = level - 1; i >= 0; i--) {**

**while (current->next[i] != nullptr && current->next[i]->value < val) {**

**current = current->next[i];**

**}**

**}**

**current = current->next[0];**

**if (current != nullptr && current->value == val) {**

**return true;**

**}**

**return false;**

**}**

**};**

**int main() {**

**SkipList skipList(4, 0.5);**

**skipList.insert(3);**

**skipList.insert(6);**

**skipList.insert(9);**

**skipList.insert(12);**

**skipList.insert(15);**

**cout << "Searching for 9: " << (skipList.search(9) ? "Found" : "Not found") << endl;**

**cout << "Searching for 7: " << (skipList.search(7) ? "Found" : "Not found") << endl;**

**return 0;**

**}**

## Thuật toán Maximal Shift

Thuật toán Maximal Shift là một thuật toán tìm kiếm chuỗi con trong một chuỗi đầu vào. Nó kết hợp giữa kỹ thuật tìm kiếm bằng cách dịch chuyển và so khớp chuỗi con. Thuật toán này tận dụng việc thực hiện dịch chuyển chuỗi con một khoảng cố định để tiến hành tìm kiếm nhanh chóng.

Thuật toán Maximal Shift hoạt động bằng cách chia chuỗi đầu vào thành các khối con, sau đó dịch chuyển chuỗi con một khoảng cố định để tìm kiếm chuỗi con. Nếu chuỗi con được tìm thấy, thuật toán trả về vị trí đầu tiên của chuỗi con trong chuỗi đầu vào. Nếu không tìm thấy, thuật toán dịch chuyển chuỗi con một khoảng cố định tiếp theo và tiếp tục quá trình tìm kiếm.

Đánh giá độ phức tạp của thuật toán:

* Độ phức tạp trong trường hợp tốt nhất: O(n/m), trong đó n là độ dài của chuỗi đầu vào và m là độ dài của chuỗi con cần tìm.
* Độ phức tạp trong trường hợp trung bình: O(n), trong đó n là độ dài của chuỗi đầu vào.
* Độ phức tạp trong trường hợp xấu nhất: O(n\*m), trong đó n là độ dài của chuỗi đầu vào và m là độ dài của chuỗi con cần tìm.

Kiểm nghiệm thuật toán Maximal Shift:

Chuỗi đầu vào: "ababcababcabc"

Chuỗi con: "abc"

Kết quả kỳ vọng: Tìm thấy chuỗi con tại vị trí 2.

Chuỗi đầu vào: "abababababab"

Chuỗi con: "aba"

Kết quả kỳ vọng: Tìm thấy chuỗi con tại các vị trí 1, 3, 5, 7 và 9.

Chuỗi đầu vào: "abcdefg"

Chuỗi con: "xyz"

Kết quả kỳ vọng: Không tìm thấy chuỗi con trong chuỗi đầu vào.

**#include <iostream>**

**#include <string>**

**using namespace std;**

**int maximalShiftSearch(const string& text, const string& pattern) {**

**int n = text.length();**

**int m = pattern.length();**

**int shift = m;**

**int i = 0;**

**while (i <= n - m) {**

**int j = m - 1;**

**while (j >= 0 && pattern[j] == text[i + j]) {**

**j--;**

**}**

**if (j < 0) {**

**return i; // Tìm thấy chuỗi con**

**} else {**

**shift = m - j - 1; // Dịch chuyển chuỗi con**

**i += shift;**

**}**

**}**

**return -1; // Không tìm thấy chuỗi con**

**}**

**int main() {**

**string text = "ababcababcabc";**

**string pattern = "abc";**

**int position = maximalShiftSearch(text, pattern);**

**if (position != -1) {**

**cout << "Pattern found at position: " << position << endl;**

**} else {**

**cout << "Pattern not found in the text." << endl;**

**}**

**return 0;**

**}**

## Thuật toán Optimal Mismatch

Thuật toán Optimal Mismatch là một thuật toán tìm kiếm chuỗi con trong một chuỗi đầu vào. Nó kết hợp giữa kỹ thuật so khớp chuỗi con và kỹ thuật đặt vị trí tối ưu cho việc so khớp. Thuật toán này được thiết kế để tìm kiếm nhanh chóng và hiệu quả trong các trường hợp mà các thuật toán tìm kiếm thông thường khác gặp khó khăn.

Thuật toán Optimal Mismatch hoạt động bằng cách so khớp chuỗi con với chuỗi đầu vào từ trái sang phải. Khi một không khớp xảy ra, thuật toán sử dụng thông tin về sự khác biệt giữa ký tự không khớp và ký tự tiếp theo của chuỗi con để đặt vị trí mới cho việc so khớp tiếp theo. Quá trình này tiếp tục cho đến khi tìm thấy chuỗi con hoặc kết thúc chuỗi đầu vào.

Đánh giá độ phức tạp của thuật toán:

* Độ phức tạp trong trường hợp tốt nhất: O(n/m), trong đó n là độ dài của chuỗi đầu vào và m là độ dài của chuỗi con cần tìm.
* Độ phức tạp trong trường hợp trung bình: O(n), trong đó n là độ dài của chuỗi đầu vào.
* Độ phức tạp trong trường hợp xấu nhất: O(n\*m), trong đó n là độ dài của chuỗi đầu vào và m là độ dài của chuỗi con cần tìm.

Kiểm nghiệm thuật toán Optimal Mismatch:

Chuỗi đầu vào: "ababcababcabc"

Chuỗi con: "abc"

Kết quả kỳ vọng: Tìm thấy chuỗi con tại vị trí 2.

Chuỗi đầu vào: "abababababab"

Chuỗi con: "aba"

Kết quả kỳ vọng: Tìm thấy chuỗi con tại các vị trí 1, 3, 5, 7 và 9.

Chuỗi đầu vào: "abcdefg"

Chuỗi con: "xyz"

Kết quả kỳ vọng: Không tìm thấy chuỗi con trong chuỗi đầu vào.

**#include <iostream>**

**#include <string>**

**using namespace std;**

**int optimalMismatchSearch(const string& text, const string& pattern) {**

**int n = text.length();**

**int m = pattern.length();**

**int i = 0;**

**while (i <= n - m) {**

**int j = m - 1;**

**while (j >= 0 && pattern[j] == text[i + j]) {**

**j--;**

**}**

**if (j < 0) {**

**return i; // Tìm thấy chuỗi con**

**} else {**

**i += max(1, j - pattern.find(text[i + j]));**

**}**

**}**

**return -1; // Không tìm thấy chuỗi con**

**}**

**int main() {**

**string text = "ababcababcabc";**

**string pattern = "abc";**

**int position = optimalMismatchSearch(text, pattern);**

**if (position != -1) {**

**cout << "Pattern found at position: " << position << endl;**

**} else {**

**cout << "Pattern not found in the text." << endl;**

**}**

**return 0;**

**}**

## Thuật toán Skip Search

Thuật toán Skip Search là một thuật toán tìm kiếm chuỗi con trong một chuỗi đầu vào. Nó kết hợp giữa kỹ thuật tìm kiếm bằng cách nhảy bước và so khớp chuỗi con. Thuật toán này tận dụng việc tạo ra một bảng chuyển đổi để xác định các bước nhảy tối ưu để tiến hành tìm kiếm nhanh chóng.

Thuật toán Skip Search hoạt động bằng cách tạo ra một bảng chuyển đổi được gọi là bảng skip. Bảng skip cho phép xác định bước nhảy tối ưu để tiến hành tìm kiếm chuỗi con. Khi so khớp chuỗi con với chuỗi đầu vào, thuật toán sẽ sử dụng bảng skip để xác định bước nhảy tiếp theo, giảm số lần so khớp cần thực hiện.

Đánh giá độ phức tạp của thuật toán:

* Độ phức tạp trong trường hợp tốt nhất: O(n/m), trong đó n là độ dài của chuỗi đầu vào và m là độ dài của chuỗi con cần tìm.
* Độ phức tạp trong trường hợp trung bình: O(n/m), trong đó n là độ dài của chuỗi đầu vào và m là độ dài của chuỗi con cần tìm.
* Độ phức tạp trong trường hợp xấu nhất: O(n), trong đó n là độ dài của chuỗi đầu vào.

Kiểm nghiệm thuật toán Skip Search:

Chuỗi đầu vào: "ababcababcabc"

Chuỗi con: "abc"

Kết quả kỳ vọng: Tìm thấy chuỗi con tại các vị trí 2, 5 và 8.

Chuỗi đầu vào: "abababababab"

Chuỗi con: "aba"

Kết quả kỳ vọng: Tìm thấy chuỗi con tại các vị trí 1, 3, 5, 7 và 9.

Chuỗi đầu vào: "abcdefg"

Chuỗi con: "xyz"

Kết quả kỳ vọng: Không tìm thấy chuỗi con trong chuỗi đầu vào.

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**vector<int> skipSearch(const string& text, const string& pattern) {**

**int n = text.length();**

**int m = pattern.length();**

**vector<int> result;**

**vector<int> skip(m, m); // Khởi tạo tất cả các giá trị trong bảng skip bằng độ dài của chuỗi con**

**for (int i = 0; i < m - 1; i++) {**

**skip[pattern[i]] = m - i - 1; // Cập nhật bước nhảy tối ưu cho các ký tự trong chuỗi con**

**}**

**int i = 0;**

**while (i <= n - m) {**

**int j = m - 1;**

**while (j >= 0 && pattern[j] == text[i + j]) {**

**j--;**

**}**

**if (j < 0) {**

**result.push\_back(i);**

**i += skip[text[i + m]];**

**} else {**

**i += skip[text[i + j]];**

**}**

**}**

**return result;**

**}**

**int main() {**

**string text = "ababcababcabc";**

**string pattern = "abc";**

**vector<int> positions = skipSearch(text, pattern);**

**if (!positions.empty()) {**

**cout << "Pattern found at positions: ";**

**for (int pos : positions) {**

**cout << pos << " ";**

**}**

**cout << endl;**

**} else {**

**cout << "Pattern not found in the text." << endl;**

**}**

**return 0;**

**}**

## Thuật toán Two way

Thuật toán Two Way (hay còn gọi là thuật toán hai chiều) là một thuật toán được sử dụng để tìm kiếm mẫu trong chuỗi. Đặc điểm nổi bật của thuật toán này là việc tận dụng cả hai hướng của chuỗi đầu vào để tăng tốc quá trình tìm kiếm. Thay vì chỉ tìm kiếm mẫu từ trái sang phải như các thuật toán tìm kiếm mẫu thông thường, thuật toán Two Way còn tìm kiếm mẫu từ phải sang trái.

Thuật toán Two Way sử dụng hai bảng chuyển tiếp để lưu trữ thông tin về sự phù hợp giữa ký tự của mẫu và chuỗi đầu vào. Bảng chuyển tiếp từ trái sang phải (forward table) lưu trữ thông tin về việc di chuyển khi ký tự không phù hợp, còn bảng chuyển tiếp từ phải sang trái (backward table) lưu trữ thông tin về việc di chuyển khi ký tự không phù hợp từ phía bên phải của mẫu.

Đánh giá độ phức tạp của thuật toán:

* Độ phức tạp trong trường hợp tốt nhất: O(n/m)
* Độ phức tạp trong trường hợp trung bình: O(n + m)
* Độ phức tạp trong trường hợp xấu nhất: O(nm), trong đó n là độ dài của chuỗi đầu vào và m là độ dài của mẫu.

Kiểm nghiệm thuật toán Two Way:

Chuỗi đầu vào: "ababcababcabc"

Mẫu: "abc"

Kết quả kỳ vọng: Tìm thấy mẫu tại các vị trí 2, 5 và 8.

Chuỗi đầu vào: "abababababab"

Mẫu: "aba"

Kết quả kỳ vọng: Tìm thấy mẫu tại các vị trí 1, 3, 5, 7 và 9.

Chuỗi đầu vào: "abcdefg"

Mẫu: "xyz"

Kết quả kỳ vọng: Không tìm thấy mẫu trong chuỗi đầu vào.

**#include <iostream>**

**#include <string>**

**using namespace std;**

**void twoWaySearch(const string& text, const string& pattern) {**

**int n = text.length();**

**int m = pattern.length();**

**int forwardTable[256];**

**int backwardTable[256];**

**// Khởi tạo bảng chuyển tiếp từ trái sang phải**

**for (int i = 0; i < 256; i++) {**

**forwardTable[i] = m;**

**}**

**for (int i = 0; i < m - 1; i++) {**

**forwardTable[pattern[i]] = m - 1 - i;**

**}**

**// Khởi tạo bảng chuyển tiếp từ phải sang trái**

**for (int i = 0; i < 256; i++) {**

**backwardTable[i] = m;**

**}**

**for (int i = m - 1; i >= 0; i--) {**

**backwardTable[pattern[i]] = i;**

**}**

**int i = 0;**

**while (i <= n - m) {**

**int j = m - 1;**

**while (j >= 0 && pattern[j] == text[i + j]) {**

**j--;**

**}**

**if (j < 0) {**

**cout << "Pattern found at position " << i << endl;**

**i += forwardTable[text[i + m]];**

**} else {**

**i += max(forwardTable[text[i + m]], backwardTable[text[i + j]]);**

**}**

**}**

**}**

**int main() {**

**string text = "ababcababcabc";**

**string pattern = "abc";**

**twoWaySearch(text, pattern);**

**return 0;**

**}**

## Thuật toán Galil-Giancarlo

Thuật toán Galil-Giancarlo là một thuật toán tìm kiếm mẫu trong chuỗi (pattern matching) được sử dụng để tìm kiếm một mẫu (pattern) trong một chuỗi văn bản (text). Thuật toán này kết hợp giữa các kỹ thuật của thuật toán Knuth-Morris-Pratt và thuật toán Boyer-Moore để cải thiện hiệu suất tìm kiếm.

Sơ lược về thuật toán Galil-Giancarlo:

* Tiền xử lý: Xây dựng các bảng chữ cái xuất hiện gần nhất (rightmost occurrence table) cho mẫu và các phần con của mẫu.
* Tìm kiếm: Bắt đầu từ vị trí cuối cùng của mẫu trong văn bản và di chuyển từ phải sang trái.
* So sánh: So sánh các ký tự từ phải sang trái của mẫu với văn bản.

1. Nếu các ký tự khớp, tiếp tục so sánh với các ký tự tiếp theo theo chiều từ phải sang trái.

* Nếu không khớp, sử dụng thông tin từ các bảng chữ cái xuất hiện gần nhất để di chuyển tới vị trí tiếp theo có thể chứa một phần của mẫu.
* Lặp lại quá trình cho đến khi tìm thấy mẫu hoặc hết văn bản.

Đánh giá độ phức tạp của thuật toán:

* Độ phức tạp trong trường hợp tốt nhất là O(n/m), trong đó n là độ dài của chuỗi văn bản và m là độ dài của mẫu. Trong trường hợp này, thuật toán chỉ cần một lần so sánh tại mỗi vị trí.
* Độ phức tạp trong trường hợp xấu nhất là O(n \* m), với n là độ dài của chuỗi văn bản và m là độ dài của mẫu. Trong trường hợp xấu nhất, tất cả các ký tự trong văn bản đều khớp với ký tự cuối cùng của mẫu, khiến thuật toán phải so sánh tại mỗi vị trí.

Kiểm nghiệm thuật toán

Text: "ABABABA"

Pattern: "ABA"

Kết quả: Tìm thấy mẫu tại các vị trí 0, 2, 4.

Text: "ABCDEF"

Pattern: "XYZ"

Kết quả: Không tìm thấy mẫu trong văn bản.

Text: "AAAAA"

Pattern: "AA"

Kết quả: Tìm thấy mẫu tại các vị trí 0, 1, 2, 3.

Lập trình theo thuật toán :

**#include <iostream>**

**#include <string>**

**#include <vector>**

**std::vector<int> galilGiancarlo(const std::string& text, const std::string& pattern) {**

**std::vector<int> occurrences;**

**int n = text.length();**

**int m = pattern.length();**

**// Xây dựng bảng chữ cái xuất hiện gần nhất**

**std::vector<int> rightmostOccurrence(256, -1);**

**for (int i = 0; i < m; i++) {**

**rightmostOccurrence[pattern[i]] = i;**

**}**

**int i = m - 1;**

**while (i < n) {**

**int j = m - 1;**

**while (j >= 0 && text[i] == pattern[j]) {**

**i--;**

**j--;**

**}**

**if (j < 0) {**

**occurrences.push\_back(i + 1);**

**i += m + 1;**

**} else {**

**int rightmost = rightmostOccurrence[text[i]];**

**if (rightmost < j) {**

**i += j - rightmost;**

**} else {**

**i += m;**

**}**

**}**

**}**

**return occurrences;**

**}**

**int main() {**

**std::string text = "ABABABA";**

**std::string pattern = "ABA";**

**std::vector<int> occurrences = galilGiancarlo(text, pattern);**

**if (occurrences.empty()) {**

**std::cout << "Pattern not found in text." << std::endl;**

**} else {**

**std::cout << "Pattern found at positions: ";**

**for (int pos : occurrences) {**

**std::cout << pos << " ";**

**}**

**std::cout << std::endl;**

**}**

**return 0;**

**}**

## Thuật toán Colussi

* Trình bày về thuật toán
  + Thuật toán Colussi là một thuật toán tìm kiếm mẫu dựa trên việc so sánh các ký tự của chuỗi và mẫu từ phải sang trái.
  + Để tối ưu hóa quá trình tìm kiếm, thuật toán Colussi sử dụng một bảng chuyển đổi để xác định các bước nhảy tối ưu khi có sự không khớp xảy ra.
  + Bước nhảy được tính dựa trên việc so sánh các ký tự không khớp và xác định vị trí tiếp theo để so sánh.
* Đánh giá độ phức tạp của thuật toán
  + Độ phức tạp thời gian của thuật toán Colussi trong trường hợp xấu nhất là O(mn), trong đó m là độ dài của mẫu và n là độ dài của chuỗi nhập.
  + Độ phức tạp không gian của thuật toán là O(m), với m là độ dài của mẫu.
* Kiểm nghiệm thuật toán
  + Testcase 1: Chuỗi và mẫu đều trống ("", ""), kết quả mong đợi là tìm thấy mẫu tại vị trí 0.
  + Testcase 2: Chuỗi trống và mẫu không trống ("", "abc"), kết quả mong đợi là không tìm thấy mẫu.
  + Testcase 3: Chuỗi không trống và mẫu trống ("abcdef", ""), kết quả mong đợi là tìm thấy mẫu tại vị trí 0.
  + Testcase 4: Chuỗi và mẫu khác nhau, không tìm thấy mẫu ("abcdef", "xyz"), kết quả mong đợi là không tìm thấy mẫu.
  + Testcase 5: Chuỗi và mẫu khớp hoàn toàn ("abcdef", "abc"), kết quả mong đợi là tìm thấy mẫu tại vị trí 0.
* Lập trình với thuật toán

**#include <iostream>**

**#include <vector>**

**#include <string>**

**using namespace std;**

**vector<int> computePrefixFunction(const string& pattern) {**

**int m = pattern.length();**

**vector<int> pi(m);**

**int k = 0;**

**for (int q = 1; q < m; q++) {**

**while (k > 0 && pattern[k] != pattern[q]) {**

**k = pi[k - 1];**

**}**

**if (pattern[k] == pattern[q]) {**

**k++;**

**}**

**pi[q] = k;**

**}**

**return pi;**

**}**

**vector<int> computeSuffixFunction(const string& pattern) {**

**int m = pattern.length();**

**vector<int> pi(m);**

**vector<int> reversedPi = computePrefixFunction(string(pattern.rbegin(), pattern.rend()));**

**for (int i = 0; i < m; i++) {**

**pi[i] = reversedPi[m - i - 1];**

**}**

**return pi;**

**}**

**void colussiSearch(const string& text, const string& pattern) {**

**int n = text.length();**

**int m = pattern.length();**

**vector<int> prefix = computePrefixFunction(pattern);**

**vector<int> suffix = computeSuffixFunction(pattern);**

**int i = 0;**

**while (i <= n - m) {**

**int j = m - 1;**

**while (j >= 0 && pattern[j] == text[i + j]) {**

**j--;**

**}**

**if (j == -1) {**

**cout << "Pattern found at index " << i << endl;**

**i += m - suffix[0];**

**} else {**

**int x = j - suffix[j];**

**int y = prefix[m - 1 - j];**

**i += max(x, y);**

**}**

**}**

**}**

**int main() {**

**string text = "This is a test string";**

**string pattern = "test";**

**colussiSearch(text, pattern);**

**return 0;**

**}**

# **Tìm kiếm mẫu từ vị trí bất kì**

## Thuật toán Smith

Thuật toán Smith là một thuật toán tìm kiếm mẫu trong chuỗi (pattern matching) được sử dụng để tìm kiếm tất cả các xuất hiện của một mẫu (pattern) trong một chuỗi văn bản (text). Thuật toán này kết hợp sử dụng hai phương pháp là tiền xử lý và tham lam (greedy) để tìm kiếm các vị trí khớp của mẫu.

Sơ lược về thuật toán Smith:

* Tiền xử lý: Xây dựng một bảng mẫu (pattern table) chứa thông tin về các vị trí của các ký tự trong mẫu. Bảng này được sử dụng để tăng tốc quá trình tìm kiếm.
* Tìm kiếm: Tại mỗi vị trí trong chuỗi văn bản, so sánh các ký tự từ phải sang trái của mẫu với văn bản.
* Nếu các ký tự tại cùng một vị trí giống nhau, tiếp tục so sánh với các ký tự tiếp theo theo chiều từ phải sang trái.
* Nếu không khớp, sử dụng thông tin từ bảng mẫu để di chuyển tới vị trí tiếp theo có thể chứa một phần của mẫu.
* Lặp lại quá trình cho đến khi tìm thấy tất cả các xuất hiện của mẫu hoặc hết văn bản.

Đánh giá độ phức tạp của thuật toán:

* Độ phức tạp trong trường hợp tốt nhất và trung bình là O(n/m), trong đó n là độ dài của chuỗi văn bản và m là độ dài của mẫu. Trong trường hợp này, thuật toán chỉ cần một lần so sánh tại mỗi vị trí.
* Độ phức tạp trong trường hợp xấu nhất là O(n \* m), với n là độ dài của chuỗi văn bản và m là độ dài của mẫu. Trong trường hợp xấu nhất, tất cả các ký tự trong văn bản đều khớp với ký tự cuối cùng của mẫu, khiến thuật toán phải so sánh tại mỗi vị trí.

Kiểm nghiệm thuật toán Smith:

Text: "ABABABA"

Pattern: "ABA"

Kết quả: Tìm thấy mẫu tại các vị trí 0, 2, 4.

Text: "ABCDEF"

Pattern: "XYZ"

Kết quả: Không tìm thấy mẫu trong văn bản.

Text: "AAAAA"

Pattern: "AA"

Kết quả: Tìm thấy mẫu tại các vị trí 0, 1, 2, 3.

Lập trình theo thuật toán:

**#include <iostream>**

**#include <string>**

**#include <vector>**

**std::vector<int> searchSmith(const std::string& text, const std::string& pattern) {**

**std::vector<int> occurrences;**

**int n = text.length();**

**int m = pattern.length();**

**for (int i = 0; i <= n - m; ) {**

**int j = m - 1;**

**while (j >= 0 && text[i + j] == pattern[j]) {**

**j--;**

**}**

**if (j < 0) {**

**occurrences.push\_back(i);**

**i++;**

**} else {**

**i += j + 1;**

**}**

**}**

**return occurrences;**

**}**

**int main() {**

**std::string text = "ABABABA";**

**std::string pattern = "ABA";**

**std::vector<int> occurrences = searchSmith(text, pattern);**

**if (occurrences.empty()) {**

**std::cout << "Pattern not found in text." << std::endl;**

**} else {**

**std::cout << "Pattern found at positions: ";**

**for (int pos : occurrences) {**

**std::cout << pos << " ";**

**}**

**std::cout << std::endl;**

**}**

**return 0;**

**}**

## Thuật toán Raita

Raita algorithm là một thuật toán tìm kiếm mẫu trong chuỗi (pattern matching) được sử dụng để tìm kiếm tất cả các xuất hiện của một mẫu (pattern) trong một chuỗi văn bản (text). Thuật toán này dựa trên việc tận dụng thông tin đã biết từ các lần tìm kiếm trước đó để tăng tốc quá trình tìm kiếm.

Sơ lược về thuật toán Raita:

* Xây dựng một bảng chứa các thông tin cần thiết để thực hiện tìm kiếm. Bảng này được gọi là bảng skip table.
* Tại mỗi vị trí trong chuỗi văn bản, so sánh mẫu với văn bản từ vị trí hiện tại.
* Nếu các ký tự tại cùng một vị trí giống nhau, tiếp tục so sánh với các ký tự tiếp theo.
* Nếu không khớp, sử dụng thông tin từ skip table để di chuyển tới vị trí tiếp theo có thể chứa một phần của mẫu.
* Lặp lại quá trình cho đến khi tìm thấy tất cả các xuất hiện của mẫu hoặc hết văn bản.

Đánh giá độ phức tạp của thuật toán:

* Độ phức tạp trong trường hợp tốt nhất là O(n/m), trong đó n là độ dài của chuỗi văn bản và m là độ dài của mẫu. Trong trường hợp này, thuật toán chỉ cần một lần so sánh tại mỗi vị trí.
* Độ phức tạp trong trường hợp trung bình và xấu nhất là O(n), với n là độ dài của chuỗi văn bản. Trong trường hợp xấu nhất, tất cả các ký tự trong văn bản đều khớp với ký tự đầu tiên của mẫu, khiến thuật toán phải so sánh tại mỗi vị trí.

Kiểm nghiệm thuật toán Raita:

Text: "ABABABA"

Pattern: "ABA"

Kết quả: Tìm thấy mẫu tại các vị trí 0, 2, 4.

Text: "ABCDEF"

Pattern: "XYZ"

Kết quả: Không tìm thấy mẫu trong văn bản.

Text: "AAAAA"

Pattern: "AA"

Kết quả: Tìm thấy mẫu tại các vị trí 0, 1, 2, 3.

Text: "ABCDABCDABCD"

Pattern: "BCD"

Kết quả: Tìm thấy mẫu tại các vị trí 1, 5, 9.

**#include <iostream>**

**#include <string>**

**#include <vector>**

**std::vector<int> buildSkipTable(const std::string& pattern) {**

**int m = pattern.length();**

**std::vector<int> skipTable(256, m);**

**for (int i = 0; i < m - 1; i++) {**

**skipTable[pattern[i]] = m - i - 1;**

**}**

**return skipTable;**

**}**

**std::vector<int> searchRaita(const std::string& text, const std::string& pattern) {**

**std::vector<int> occurrences;**

**int n = text.length();**

**int m = pattern.length();**

**std::vector<int> skipTable = buildSkipTable(pattern);**

**int i = m - 1;**

**while (i < n) {**

**int j = m - 1;**

**while (j >= 0 && text[i] == pattern[j]) {**

**i--;**

**j--;**

**}**

**if (j < 0) {**

**occurrences.push\_back(i + 1);**

**i += m + 1;**

**} else {**

**i += skipTable[text[i]];**

**}**

**}**

**return occurrences;**

**}**

**int main() {**

**std::string text = "ABABABA";**

**std::string pattern = "ABA";**

**std::vector<int> occurrences = searchRaita(text, pattern);**

**if (occurrences.empty()) {**

**std::cout << "Pattern not found in text." << std::endl;**

**} else {**

**std::cout << "Pattern found at positions: ";**

**for (int pos : occurrences) {**

**std::cout << pos << " ";**

**}**

**std::cout << std::endl;**

**}**

**return 0;**

**}**

## Thuật toán Apostolico-Giancarlo

Thuật toán Apostolico-Giancarlo (AG) là một thuật toán tìm kiếm mẫu trong chuỗi (pattern matching) được sử dụng để tìm kiếm một mẫu (pattern) trong một chuỗi văn bản (text). Thuật toán này kết hợp giữa thuật toán Knuth-Morris-Pratt và thuật toán Boyer-Moore để cải thiện hiệu suất tìm kiếm.

Sơ lược về thuật toán Apostolico-Giancarlo:

* Tiền xử lý: Xây dựng bảng chữ cái xuất hiện gần nhất (rightmost occurrence table) cho mẫu. Bảng này chứa thông tin về vị trí xuất hiện gần nhất của mỗi ký tự trong mẫu.
* Tìm kiếm: Bắt đầu từ vị trí cuối cùng của mẫu trong văn bản và di chuyển từ phải sang trái.
* So sánh: So sánh các ký tự từ phải sang trái của mẫu với văn bản.
* Nếu các ký tự khớp, tiếp tục so sánh với các ký tự tiếp theo theo chiều từ phải sang trái.
* Nếu không khớp, sử dụng thông tin từ bảng chữ cái xuất hiện gần nhất để di chuyển tới vị trí tiếp theo có thể chứa một phần của mẫu.
* Lặp lại quá trình cho đến khi tìm thấy mẫu hoặc hết văn bản.

Đánh giá độ phức tạp của thuật toán:

* Độ phức tạp trong trường hợp tốt nhất là O(n/m), trong đó n là độ dài của chuỗi văn bản và m là độ dài của mẫu. Trong trường hợp này, thuật toán chỉ cần một lần so sánh tại mỗi vị trí.
* Độ phức tạp trong trường hợp xấu nhất là O(n \* m), với n là độ dài của chuỗi văn bản và m là độ dài của mẫu. Trong trường hợp xấu nhất, tất cả các ký tự trong văn bản đều khớp với ký tự cuối cùng của mẫu, khiến thuật toán phải so sánh tại mỗi vị trí.

Kiểm nghiệm thuật toán:

Text: "ABABABA"

Pattern: "ABA"

Kết quả: Tìm thấy mẫu tại các vị trí 2, 4.

Text: "ABCDEF"

Pattern: "XYZ"

Kết quả: Không tìm thấy mẫu trong văn bản.

Text: "AAAAA"

Pattern: "AA"

Kết quả: Tìm thấy mẫu tại các vị trí 0, 1, 2, 3.

Lập trình theo thuật toán :

**#include <iostream>**

**#include <string>**

**#include <vector>**

**std::vector<int> apostolicoGiancarlo(const std::string& text, const std::string& pattern) {**

**std::vector<int> occurrences;**

**int n = text.length();**

**int m = pattern.length();**

**// Xây dựng bảng chữ cái xuất hiện gần nhất**

**std::vector<int> rightmostOccurrence(256, -1);**

**for (int i = 0; i < m; i++) {**

**rightmostOccurrence[pattern[i]] = i;**

**}**

**int i = m - 1;**

**while (i < n) {**

**int j = m - 1;**

**while (j >= 0 && text[i] == pattern[j]) {**

**i--;**

**j--;**

**}**

**if (j < 0) {**

**occurrences.push\_back(i + 1);**

**i += m + 1;**

**} else {**

**int rightmost = rightmostOccurrence[text[i]];**

**if (rightmost < j) {**

**i += j - rightmost;**

**} else {**

**i += m;**

**}**

**}**

**}**

**return occurrences;**

**}**

**int main() {**

**std::string text = "ABABABA";**

**std::string pattern = "ABA";**

**std::vector<int> occurrences = apostolicoGiancarlo(text, pattern);**

**if (occurrences.empty()) {**

**std::cout << "Pattern not found in text." << std::endl;**

**} else {**

**std::cout << "Pattern found at positions: ";**

**for (int pos : occurrences) {**

**std::cout << pos << " ";**

**}**

**std::cout << std::endl;**

**}**

**return 0;**

**}**

## Thuật toán Horspool

* Trình bày về thuật toán

Thuật toán Horspool được sử dụng để tìm kiếm mẫu (pattern) trong văn bản (text). Thuật toán này dựa trên việc thực hiện các phép dịch và phép nhảy để tăng tốc quá trình tìm kiếm. Ý tưởng chính của thuật toán là sử dụng một bảng dịch (shift table) để xác định khoảng cách cần dịch chuyển khi không khớp ký tự cuối cùng của mẫu với văn bản.

Thuật toán Horspool hoạt động theo các bước sau:

* Tạo bảng dịch: Đối với mỗi ký tự có thể xuất hiện trong mẫu, tính toán khoảng cách từ ký tự đó đến ký tự cuối cùng trong mẫu. Nếu ký tự không xuất hiện trong mẫu, khoảng cách sẽ là độ dài của mẫu.
* Duyệt qua văn bản từ trái sang phải, so sánh ký tự cuối cùng của mẫu với ký tự tại vị trí tương ứng trong văn bản.
* Nếu ký tự cuối cùng của mẫu không khớp với ký tự trong văn bản, sử dụng bảng dịch để xác định khoảng cách cần dịch chuyển. Dịch chuyển văn bản sang phải theo khoảng cách đã xác định.
* Nếu ký tự cuối cùng của mẫu khớp với ký tự trong văn bản, tiến hành so sánh các ký tự còn lại của mẫu từ phải sang trái.
* Nếu tất cả các ký tự trong mẫu khớp với các ký tự tương ứng trong văn bản, tìm thấy một vị trí xuất hiện của mẫu trong văn bản.

Đánh giá độ phức tạp của thuật toán

* Độ phức tạp thời gian của thuật toán Horspool là O(n + m), trong đó n là độ dài của văn bản và m là độ dài của mẫu.
* Độ phức tạp không gian của thuật toán là O(k), trong đó k là số lượng ký tự có thể xuất hiện trong mẫu.

Kiểm nghiệm thuật toán

Testcase 1:

Text: "This is a test string."

Pattern: "test"

Kết quả mong đợi: Tìm thấy mẫu "test" tại vị trí 10 trong văn bản.

Testcase 2:

Text: "ABCABCABCABC"

Pattern: "BCA"

Kết quả mong đợi: Tìm thấy mẫu "BCA" tại vị trí 5 và 9 trong văn bản.

Testcase 3:

Text: "Hello World"

Pattern: "Foo"

Kết quả mong đợi: Không tìm thấy mẫu "Foo" trong văn bản.

* Lập trình với thuật toán

**#include <iostream>**

**#include <vector>**

**#include <string>**

**std::vector<int> build\_shift\_table(const std::string& pattern) {**

**std::vector<int> table(256, pattern.length());**

**for (int i = 0; i < pattern.length() - 1; i++) {**

**table[pattern[i]] = pattern.length() - 1 - i;**

**}**

**return table;**

**}**

**std::vector<int> horspool(const std::string& text, const std::string& pattern) {**

**std::vector<int> matches;**

**int n = text.length();**

**int m = pattern.length();**

**std::vector<int> shift\_table = build\_shift\_table(pattern);**

**int i = m - 1;**

**while (i < n) {**

**int k = 0;**

**while (k < m && pattern[m - 1 - k] == text[i - k]) {**

**k++;**

**}**

**if (k == m) {**

**matches.push\_back(i - m + 1);**

**}**

**i += shift\_table[text[i]];**

**}**

**return matches;**

**}**

**int main() {**

**std::string text = "This is a test string.";**

**std::string pattern = "test";**

**std::vector<int> matches = horspool(text, pattern);**

**if (matches.empty()) {**

**std::cout << "Pattern not found." << std::endl;**

**} else {**

**std::cout << "Pattern found at positions: ";**

**for (int i = 0; i < matches.size(); i++) {**

**std::cout << matches[i] << " ";**

**}**

**std::cout << std::endl;**

**}**

**return 0;**

**}**

# **TỔNG KẾT**

Chuyên đề công nghệ phần mềm là một lĩnh vực quan trọng trong lĩnh vực công nghệ thông tin, nơi mà chúng ta tìm hiểu về quy trình và phương pháp để phát triển, triển khai và duy trì phần mềm. Trong chuyên đề này, các thuật toán đóng vai trò quan trọng trong việc giải quyết các vấn đề liên quan đến phần mềm, từ việc tìm kiếm mẫu trong chuỗi, xử lý văn bản, đến phân tích dữ liệu và tối ưu hóa thuật toán.

Có nhiều thuật toán khác nhau trong chuyên đề công nghệ phần mềm, mỗi thuật toán có ứng dụng và đặc điểm riêng. Đánh giá độ phức tạp của mỗi thuật toán giúp chúng ta hiểu về hiệu suất và thời gian chạy của thuật toán. Việc lựa chọn và áp dụng đúng thuật toán phù hợp với bài toán cụ thể là rất quan trọng để đạt được hiệu quả tối ưu trong phát triển phần mềm.

Việc thực hiện các testcase cho các thuật toán là một phần không thể thiếu trong quá trình kiểm thử và xác nhận tính đúng đắn của thuật toán. Viết các testcase có thể giúp chúng ta kiểm tra và đánh giá tính chính xác, hiệu suất và độ tin cậy của thuật toán.

Tóm lại, các thuật toán trong chuyên đề công nghệ phần mềm đóng vai trò quan trọng trong việc xử lý và giải quyết các vấn đề liên quan đến phần mềm. Việc hiểu về các thuật toán này và khả năng áp dụng chúng vào thực tế sẽ giúp chúng ta phát triển phần mềm hiệu quả, tối ưu hóa hiệu suất và cải thiện trải nghiệm người dùng.