1. 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: ""

Giải thích: 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;**

**}**

1. 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;**

**}**

1. **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;**

**}**

1. 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;**

**}**

1. 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

1. 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;**

**}**

1. 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

**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: "abababcababcabcabc"**

**Pattern: "abc"**

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

**Testcase 3:**

**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;**

**}**

1. Thuật toán

* Trình bày về thuật toán
* Đánh giá độ phức tạp của thuật toán
* Kiểm nghiệm thuật toán
* Lập trình với thuật toán

1. Thuật toán

* Trình bày về thuật toán
* Đánh giá độ phức tạp của thuật toán
* Kiểm nghiệm thuật toán
* Lập trình với thuật toán