CÁC PHƯƠNG PHÁP KIỂM TRA SAO CHÉP BÀI TẬP LẬP TRÌNH của sinh viên TRONG TRƯỜNG ĐẠI HỌC

PLAGIARISM detection METHODS IN STUDENTS’ CODE IN THE UNIVERSITY

SVTH: Nguyễn Tấn Bảo, Tăng Văn Quốc Chí

Lớp 13T2, 14T2, Khoa Công nghệ thông tin, Trường Đại học Bách Khoa, Đại học Đà Nẵng

Email: tanbaonguyen95@gmail.com, tvquocchi@gmail.com

GVHD: TS. Phạm Minh Tuấn

Khoa Công nghệ thông tin, Trường Đại học Bách Khoa, Đại học Đà Nẵng

Email: pmtuan@dut.udn.vn

**Tóm tắt –** Việc sao chép lẫn nhau luôn là vấn đề lớn trong thi cử, đặc biệt là trong các môn học thuộc khối ngành Công nghệ thông tin và lập trình. Sinh viên chỉ cần một vài thao tác đơn giản là có thể sao chép được bài giải của một sinh viên khác. Vì vậy, việc phát hiện sao chép các bài tập lập trình trở nên là một vấn đề cấp thiết. Bài báo này sẽ nghiên cứu các cách thức so sánh hai mã nguồn bài tập lập trình trước đây, đưa ra những nhược điểm của các phương pháp này. Qua đó bài báo đề xuất các cách thức xử lý hiệu quả hơn thông qua tiền xử lý, áp dụng mảng hậu tố và nén xâu ký tự. Cuối cùng, bài báo sẽ đưa ra bốn phương pháp phát hiện sao chép để thử nghiệm, so sánh độ hiệu quả của từng phương pháp và xác định hướng đi tốt nhất.

**Từ khóa -** So sánh xâu, tiền xử lý văn bản, phát hiện sao chép, mảng hậu tố, nén xâu

**Abstract –** Plagiarism is one of the most fundamental problems in testing the abilities of students, especially in subjects that belong to Information Technology and Programming area. One student can perform some trivial tasks to copy code of another student. Therefore, detecting plagiarism in students’ code remains a massive problem. This paper researches on conventional methods in comparing students’ code, and the drawbacks of these techniques. Subsequently, this paper proposes more efficient methods through preprocessing, applying suffix array and hashing string to increase the effective of the plagiarism detection. In conclusion, this paper presents four plagiarism detection methods, compares these approaches in terms of efficiency and finds out the most effective method.

**Key words –** Comparing strings, preprocessing strings, detecting plagiarism, suffix array, hashing

# Đặt vấn đề

Ngày nay, cùng với sự phát triển nhanh chóng của máy tính điện tử, sinh viên, đặc biệt là sinh viên thuộc các ngành Công nghệ thông tin, được tiếp cận nhiều hơn đến việc lập trình trên máy tính. Nhiều học phần được thiết kế linh động, sinh viên hiện nay có thể lập trình rồi gửi bài làm của mình cho giáo viên để chạy và chấm điểm. Các hệ thống chấm [1][2] cũng có thể được thiết kế tự động, tiết kiệm thời gian cho cả sinh viên và giảng viên.

Việc so sánh bài làm của hai sinh viên xem có hoàn toàn giống nhau hay không là một việc rất dễ dàng với nhiều công cụ hỗ trợ. Tuy nhiên, với các bài chép nhau một phần, việc phát hiện sẽ vô cùng khó khăn. Điều tương tự cũng xảy ra trong trường hợp những sinh viên sao chép nhưng cố ý thay đổi hoặc thêm vào những yếu tố không cần thiết vào bài làm của mình, dù cấu trúc hoàn toàn giống nhau để qua mắt giảng viên.

Bài báo này tiến hành việc nghiên cứu về các phương pháp so sánh độ tương đồng của các bài tập lập trình của sinh viên và đưa ra đề xuất phương pháp hợp lý nhất.

# Các phương pháp liên quan trước đây

## Thuật toán so khớp hai đoạn chương trình

Trong các phương pháp trước đây, khi muốn so sánh hai đoạn chương trình với nhau, người ta thường chuyển mỗi đoạn chương trình về thành một xâu thống nhất, sau đó so sánh hai xâu này với nhau bằng phép so sánh có sẵn trong các ngôn ngữ lập trình xem hai xâu này có giống nhau hay không.

|  |
| --- |
| function compare() {  string A = readFileA();  // đọc từng dòng của tập tin A và chắp // lại thành một xâu  string B = readFileB();  // đọc từng dòng của tập tin B và chắp // lại thành 1 xâu  if (A == B) return true;  // nếu xâu A và B giống nhau thì khẳng // định là sao chép 100%  return false;  // ngược lại thì không sao chép  } |

Tuy nhiên, phương pháp so khớp trên chỉ có thể phát hiện ra hai xâu có giống nhau toàn bộ hay là không. Người sao chép chỉ cần tinh ý sửa một chút trong đoạn chương trình mình vừa sao chép thì sẽ không bị hệ thống phát hiện ra.

## Thuật toán Edit Distance

Một phương pháp tốt hơn được sử dụng trước đây là thuật toán Edit Distance [3].

Edit Distance là một phương pháp để kiểm tra xem hai xâu kí tự khác nhau như thế nào bằng cách đếm số phép biến đổi để từ xâu này biến thành xâu kia. Cụ thể là các phép biến đổi sau:

* Đổi một ký tự X thành một ký tự Y khác.
* Xóa đi một ký tự.
* Thêm một ký tự bất kỳ vào một vị trí bất kỳ.

Nhờ vào phương pháp này, chúng ta có thể xác định được số các phép biến đổi để biến văn bản này thành văn bản kia, hay cụ thể trong trường hợp của chúng ta là từ đoạn mã này sang đoạn mã kia. Từ đó, chúng ta sẽ suy ra được đoạn mã này sao chép đoạn mã kia bao nhiêu phần trăm.

Tư tưởng của phương pháp này là quy hoạch động (Dynamic Programming). Cho hai xâu ban đầu là và . Gọi mảng hai chiều là số phép biến đổi để biến xâu thành xâu . Ta có các trường hợp cơ bản như sau:

* (hai xâu rỗng thì không cần phép biến đổi nào).
* (cần xóa đi ký tự của xâu để nó thành xâu rỗng với ).
* (cần thêm vào xâu A ký tự để nó thành xâu với ).

Đến đây, chúng ta chỉ cần duyệt như sau, với mọi và ta có,

với:

* (khi bài toán quy về số phép biến đổi để xâu thành xâu , vì và đã giống nhau rồi, còn khi thì cần phải thực hiện thao tác biến đổi thành ).
* (bài toán quy về số phép biến đổi để xâu thành xâu khi thực hiện thao tác thêm một ký tự giống ký tự vào sau xâu ).
* (bài toán quy về số phép biến đổi để xâu thành xâu khi thực hiện thao tác xóa ký tự ).

Kết quả cuối cùng chính là giá trị . Từ đây, ta có thể lấy được tỉ lệ sao chép giữa hai đoạn chương trình dựa theo số phép biến đổi từ xâu A sang xâu B. Độ phức tạp của thuật này là .

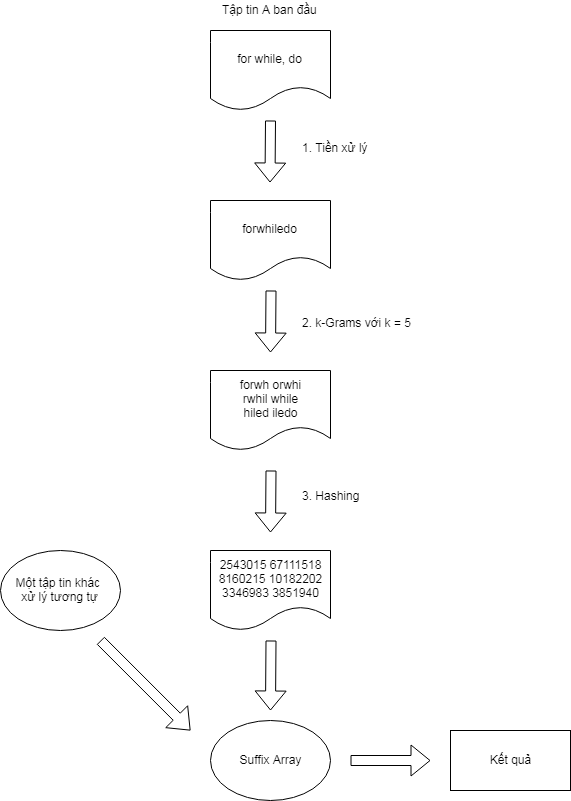
# Phương pháp đề xuất

## Tiêu chí

Trong nghiên cứu này, hai mã nguồn được xem là sao chép lẫn nhau khi cấu trúc cơ bản của chương trình giống nhau, đổi tên các biến, thêm các thư viện, khai báo thừa. Ngoài ra, mã nguồn sao chép có thể thay đổi vị trí các hàm sao cho khác với mã nguồn gốc. Còn với các trường hợp can thiệp quá nhiều, ví dụ như đổi vòng lặp for thành vòng lặp while thì không được xem là sao chép vì người sao chép trong trường hợp này đã hiểu về bản chất của mã nguồn và các cú pháp liên quan.

## Sơ đồ khối của phương pháp

Hình 1 cho chúng ta thấy sơ đồ khối của phương pháp đề xuất.



**Hình 1:** Sơ đồ khối quy trình đề xuất để kiểm tra sao chép

## Tiền xử lý sơ bộ

Để không bị hệ thống phát hiện ra việc sao chép, người sao chép chỉ cần thêm các kí tự thừa nhưng không làm ảnh hưởng đến cấu trúc chương trình, ví dụ như các kí tự trống, tab, xuống dòng hoặc các dòng chú thích (comment). Khi đó, xâu tạo ra từ chương trình sẽ thay đổi rất nhiều, gây khó khăn cho quá trình kiểm tra độ tương đồng.

Như vậy, cần thiết phải xử lý xâu được tạo ra bởi chương trình trước khi so sánh. Vì thế, chúng ta có thể tiền xử lý để bỏ đi các kí tự thừa nêu trên và thực hiện phép so sánh. Thuật toán tiền xử lý được đề xuất như sau:

|  |
| --- |
| function preProcess() {  compileFileA();  // biên dịch sơ bộ file để bỏ hết các // chú thích trong mã nguồn  string A = readFileA();  // đọc từng dòng của tập tin A và chắp // lại thành một xâu  string res;  // xâu sau khi xử lý  for (all character in string A) {  if (A[i] == ‘ ’) continue;  if (A[i] == ‘\t’) continue;  if (A[i] == ‘\n’) continue;  // bỏ hết dấu cách, tab và enter  res += A[i];  }  return res;  } |

## Phương pháp tiền xử lý đề xuất

Tuy nhiên, tiền xử lý sơ bộ không thể phát hiện ra các mã nguồn sao chép bằng cách khai báo nhiều thư viện, định nghĩa những kiểu dữ liệu dư thừa, hoặc đổi tên biến. Vì thế, chúng ta cần xử lý thêm các dữ kiện đó. Ví dụ với mã nguồn C++, chúng ta cần thực hiện các thao tác sau:

* Xóa bỏ tất cả các khai báo thư viện (include).
* Xóa các khai báo dư thừa và gần như có trong tất cả các mã nguồn bài tập, ví dụ như “using namespace std”.
* Thay các chỗ “typedef” trong mã nguồn bằng “#define” để giúp cho trình biên dịch tự động thay thế các chỗ định nghĩa kiểu dữ liệu.

Sau đó, chúng ta cho trình biên dịch dịch sơ bộ (preprocessor in compiler) mã nguồn để nó thay thế các kiểu dữ liệu mà ta định nghĩa thành kiểu dữ liệu gốc và xóa đi các dòng chú thích. Bước tiếp theo trong tiền xử lý chính là việc thay thế tên các biến trong mã nguồn thành chính kiểu dữ liệu được định nghĩa cho biến đó, đồng thời bỏ đi các khai báo biến. Việc này giúp cho việc đổi tên biến trở nên không hiệu quả khi tất cả tên biến sẽ hiển thị dưới dạng tên kiểu dữ liệu. Quá trình này đòi hỏi một tư duy tốt về xử lý ngôn ngữ tự nhiên, cụ thể là ngôn ngữ lập trình và nắm rõ cú pháp của ngôn ngữ lập trình đó. Cuối cùng, chúng ta chỉ cần xóa thêm các dấu cách, dấu tab và dấu xuống dòng. Như vậy quá trình tiền xử lý đã hoàn thành.

## Phương pháp -grams và thuật toán Rabin-Karp

Một điểm yếu của Edit Distance đó chính là việc phát hiện ra những đoạn giống nhau quá ngắn, kể cả các kí tự rời rạc cũng được xem là sao chép. Để khắc phục nhược điểm này, phương pháp -grams [4] được sử dụng.

Một -gram là một xâu con có độ dài là . Nếu như đoạn mã nguồn có độ dài là thì mã nguồn đó có -gram trong đó. Ví dụ cho một đoạn mã nguồn như sau:

|  |
| --- |
| for while do do, do for do |

Sau khi tiền xử lý, ta có xâu sau với độ dài là 19 kí tự:

|  |
| --- |
| forwhiledododofordo |

Sử dụng phương pháp -Grams với , ta có các -gram của xâu trên bao gồm:

|  |
| --- |
| forwh orwhi rwhil while hiled iledo ledod edodo dodod ododo dodof odofo dofor oford fordo |

Như vậy ta sẽ có số -gram trong xâu là -gram. Sau khi có được các -gram trong mỗi mã nguồn, chỉ cần có một -gram trong mã nguồn này xuất hiện trong mã nguồn kia nghĩa là đã phát hiện sao chép. Phương pháp này đảm bảo độ dài đoạn mã bị chép là đủ dài để được xem là đã sao chép.

Tuy nhiên, khi giá trị lớn, việc so sánh 2 xâu -gram trong hai mã nguồn có độ phức tạp là , rất tốn thời gian khi tăng giá trị lên. Chúng ta cần phải nén xâu lại thành một giá trị để so sánh cho tiện lợi. Thuật toán Rabin-Karp (hay còn gọi là Hashing) [4] sẽ giải quyết vấn đề đó.

Tư tưởng của Rabin-Karp như sau, gọi lần lượt là các phần tử trong một -gram, các phần tử này được biểu diễn dưới dạng số theo một hệ cơ số nào đó. Để cho đơn giản, các phần tử sẽ được lấy theo mã ASCII của các kí tự. Thì hash của -gram này sẽ được biểu diễn như sau

Để biểu diễn đoạn tiếp theo , chúng ta chỉ cần trừ đi , nhân tất cả với sau đó cộng

Ví dụ, với -gram “forwh”, ta sẽ Hash nó với (26 kí tự trong bảng chữ cái), với kí tự tương ứng với 0, kí tự tương ứng với 1,…, kí tự tương ứng với 25. Như vậy:

Tính tiếp cho ta có:

Giá trị xâu trở về thành số. Bằng cách này, chúng ta có thể so sánh các -gram với độ phức tạp là , việc thực thi tăng lên rất nhiều.

## Suffix Array

Sau khi đã thực hiện việc Hashing mã nguồn, chúng ta cần một thuật toán để tìm một phần của mảng Hashing này trong mảng Hashing kia. Thuật toán được đề xuất ở đây là Suffix Array [5].

Suffix Array – mảng hậu tố, đúng như tên gọi của nó, là một thuật toán để sắp xếp các hậu tố của một xâu đầu vào theo thứ tự từ điển. Đây đồng thời cũng là cách nhanh nhất để tìm kiếm một phần của xâu này trong một xâu khác.

Ví dụ, cho một xâu , gọi là vị trí trong xâu (bắt đầu từ 0), là mảng chứa vị trí bắt đầu của các xâu hậu tố của . Từ mảng , chúng ta có thể tạo được mảng (Longest Common Prefix) với ý nghĩa rằng nó là giá trị độ dài lớn nhất mà hậu tố thứ trong mảng trùng với hậu tố thứ trong mảng .

Kết quả sau khi dùng Suffix Array như sau:

**Bảng 1:** Kết quả Suffix Array cho xâu

|  |  |  |  |
| --- | --- | --- | --- |
| **i** |  |  | **Hậu tố** |
| 0 | 7 | 0 | A |
| 1 | 5 | 1 | ACA |
| 2 | 3 | 1 | AGACA |
| 3 | 1 | 1 | ATAGACA |
| 4 | 6 | 0 | CA |
| 5 | 4 | 0 | GACA |
| 6 | 0 | 2 | GATAGACA |
| 7 | 2 | 0 | TAGACA |

Có nhiều cách khác nhau để cài đặt Suffix Array, với các độ phức tạp khác nhau, từ đến . Cài đặt Suffix Array đơn giản nhất với độ phức tạp như sau:

|  |
| --- |
| bool cmp(int a, int b) {  return strcmp(S+a, S+b) < 0;  } // so sánh trong O(n)  int main() {  n = strlen(S)  for (i : 0 -> n-1) SA[i] = i;  // khởi trị SA  sort(SA, SA+n, cmp);  // sắp xếp mảng SA bằng hàm so sánh cmp  } |

Việc tính mảng cũng có nhiều cách, một cách đơn giản nhất với độ phức tạp như sau:

|  |
| --- |
| function computeLCP() {  LCP[0] = 0;  for (i : 0 -> n-1) {  int L = 0;  while (S[SA[i]+L] == S[SA[i-1]+L]) L++;  // L là độ dài tiền tố dài nhất // của hậu tố SA[i] so với hậu tố // SA[i-1]  LCP[i] = L;  }  } |

Như vậy, để tìm một phần của xâu này trong xâu kia, ta chỉ việc ghép hai xâu với nhau và dùng Suffix Array trên xâu chung.

Ví dụ xâu , xâu , vậy xâu chung là , hai kí tự và nằm sau các xâu và để neo lại các giá trị , tránh các giá trị tràn qua xâu kia.

Vậy kết quả khi dùng Suffix Array như sau:

**Bảng 2:** Kết quả Suffix Array cho xâu

|  |  |  |  |
| --- | --- | --- | --- |
| **i** |  |  | **Hậu tố** |
| 0 | 12 | 0 | # |
| 1 | 8 | 0 | $AGA# |
| 2 | 11 | 0 | A# |
| 3 | 7 | 1 | A$AGA# |
| 4 | 5 | 1 | ACA$AGA# |
| 5 | 9 | 1 | AGA# |
| 6 | 3 | 3 | AGACA$AGA# |
| 7 | 1 | 1 | ATAGACA$AGA# |
| 8 | 6 | 0 | CA$AGA# |
| 9 | 10 | 0 | GA# |
| 10 | 4 | 2 | GACA$AGA# |
| 11 | 0 | 2 | GATAGACA$AGA# |
| 12 | 2 | 0 | TAGACA$AGA# |

Đến đây, ta khảo sát các cụm có , đó là các đoạn từ 2 đến 7 và từ 9 đến 11 (tính luôn các ngay trước đoạn). Xét riêng đoạn từ 2 đến 7, ta thấy kí tự thứ 3 của xâu sẽ trùng với các kí tự thứ 2, 4, 6, 8 của xâu , và trùng với cả kí tự đầu tiên của chính xâu . Ta cũng có thể tìm ra đoạn của xâu trùng với đoạn của xâu S thông qua .

Suffix Array là một cách hữu hiệu để tìm ra tất cả các đoạn trùng của xâu này với xâu kia. Nghiên cứu này sẽ dùng mỗi giá trị của Hashing như một phần tử của xâu kí tự và áp dụng Suffix Array lên mảng số Hashing, thay vì xâu kí tự.

# Kết quả nghiên cứu và khảo sát

Nghiên cứu này triển khai hệ thống lập trình online, với sự tham gia của 66 người dùng. Nghiên cứu tập trung thực nghiệm kiểm tra sao chép trong bộ dữ liệu của trang web chấm bài trực tuyến của Đại học Bách khoa – Đại học Đà Nẵng (Online Judge – bkdnOJ, URL: bkdnoj.dut.udn.vn), dữ liệu bao gồm 295 mã nguồn đã được nộp bằng ngôn ngữ C++ lên trang web này, thuộc 40 bài tập và tất cả các mã nguồn này đều ở trạng thái Accepted đối với tất cả các testcase của các bài tập. Nghiên cứu thiết lập tỉ lệ để được coi như đã phát hiện sao chép của tất cả các phương pháp là 80%. Trong số 295 mã nguồn này có một mã nguồn được cố ý tạo thêm lấy từ một mã nguồn có trong các mã nguồn có sẵn và thay đổi các tên biến, thứ tự hàm cũng như các thư viện được khai báo thêm nhưng không làm thay đổi cấu trúc chương trình. Mã nguồn này khá dài (108 dòng) có nhiều hàm và biến, phù hợp với việc nghiên cứu. Khi nghiên cứu, mức độ sao chép sẽ được tính theo phần trăm và sắp xếp từ tỉ lệ lớn nhất xuống bé nhất, cuối cùng lấy thứ hạng của cặp mã nguồn được cài thêm để so sánh độ hiệu quả giữa các phương pháp.

Việc nghiên cứu so sánh tương đồng giữa 2 mã nguồn bài tập sẽ dựa trên 4 phương pháp:

* Tiền xử lý sơ bộ ở mục **3.1.** (bỏ dấu cách, dấu tab, chú thích…) và so sánh hai mã nguồn bằng phương pháp cổ điển.
* Tiền xử lý sơ bộ ở mục **3.1** và dùng Edit Distance để so sánh hai mã nguồn.
* Tiền xử lý đề xuất ở mục **3.2** và dùng Edit Distance để so sánh hai mã nguồn.
* Tiền xử lý đề xuất ở mục **3.2** và dùng k-Grams với k = 15, Hashing kết hợp với Suffix Array để so sánh hai mã nguồn.

**Bảng 3:** Kết quả số cặp mã nguồn phát hiện sao chép theo phần trăm sao chép và thứ hạng của mã nguồn cài thêm trong mỗi phương pháp

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **PP** | **80-<85%** | **85-<90%** | **90-<95%** | **95-<100%** | **100%** | **Tổng** | **Thứ hạng của mã nguồn cài thêm** |
| 1 | 0 | 0 | 0 | 0 | 8 | 8 | Không có hạng |
| 2 | 8 | 5 | 2 | 0 | 8 | 23 | Không có hạng |
| 3 | 23 | 21 | 8 | 4 | 13 | 69 | 68 (80.35%) |
| 4 | 7 | 3 | 7 | 4 | 13 | 34 | 14 (96.83%) |

Đối với phương pháp đầu tiên, phương pháp này phát hiện được 8 cặp mã nguồn sao chép nhau trong tổng số 295 mã nguồn. Các mã nguồn này sao chép giống hệt nhau nhưng có thêm bớt một số chi tiết thừa như xuống dòng, dấu tab hoặc chú thích.

Với phương pháp thứ hai, phương pháp này phát hiện 23 cặp mã nguồn sao chép. 23 cặp mã nguồn này có chứa 8 cặp mã nguồn ở phương pháp đầu tiên. Mã nguồn được cài thêm vào không xuất hiện trong danh sách này. Số lượng các mã nguồn có độ tương đồng cao (trên 80%) cũng tăng lên.

Với phương pháp thứ ba, với việc áp dụng tiền xử lý đề xuất, số các cặp mã nguồn có độ tương đồng cao tăng lên đáng kể. Mã nguồn được cài thêm vào cũng xuất hiện ở vị trí gần cuối trong danh sách với độ tương đồng là 80.35%. Điều này có nghĩa là tiền xử lý mạnh hơn có thể phát hiện được sao chép.

Với phương pháp thứ tư, số lượng các cặp mã nguồn có độ tương đồng cao lại bị giảm đi, nhưng mã nguồn được cài thêm lại tăng độ tương đồng lên gần tối đa (96.83%). Điều này có nghĩa là các cặp mã nguồn thực sự sao chép nhau một đoạn đủ dài thì tăng độ tương đồng, còn các cặp mã nguồn chỉ sao chép các đoạn rời rạc và đoạn nhỏ thì giảm độ tương đồng. Việc mã nguồn cài thêm tăng độ tương đồng cũng cho thấy điểm yếu của Edit Distance trong việc so sánh hai mã nguồn. Nếu mã nguồn dài thì dùng phương pháp thứ tư này thực sự rất hiệu quả. Tuy nhiên, Edit Distance cũng cho thấy độ hiệu quả nhất định trong việc so sánh những mã nguồn ngắn.

# Kết luận

Các phương pháp đề xuất về tiền xử lý và Hashing kết hợp với Suffix Array tỏ ra rất hiệu quả trong việc phát hiện sao chép giữa hai mã nguồn. Tuy việc tiền xử lý chỉ được nghiên cứu và áp dụng chỉ với mã nguồn ngôn ngữ C++ nhưng đã cho thấy tiềm năng rất lớn trong lĩnh vực này.

**Tài liệu tham khảo**

1. M. Mirzayanov. Codeforces. M. Mirzayanov. 2018. URL: <http://codeforces.com>
2. Directi Group. Codechef. Directi. 2018. URL: <https://www.codechef.com>
3. Navarro, Gonzalo (1 March 2001). A guided tour to approximate string matching. ACM Computing Surveys. 33 (1): 31–88. [doi:10.1145/375360.375365](https://doi.org/10.1145%2F375360.375365)
4. S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In ACM, editor, Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003, San Diego, California, June 09–12, 2003, pages 76–85, New York, NY 10036, USA, 2003. ACM Press.
5. Manber, U. and Myers, G. (1990). Suffix arrays: a new method for on-line string searches. In Proc. 1st Annual ACM-SIAM Symp. on Discrete Algorithms, pages 319–327.

Phụ lục

Mã nguồn C++ được cài thêm vào để thử nghiệm như sau:

|  |
| --- |
| #include <bits/stdc++.h>  #include <iostream>  #include <string>  #include <vector>  #include <map>  #include <algorithm>  #include <set>  #include <cmath>  #define X first  #define Y second  typedef long long ll;  #define FOR(i,a,b) for(ll i = a;i<=b;i++)  #define FOD(i,a,b) for(ll i = a;i>=b;i--)  #define pb push\_back  #define mp make\_pair  using namespace std;  typedef pair<ll,ll>ii;  typedef priority\_queue<ll,vector<ll>,greater<ll> > heap\_min;  const ll pp = 1e6+5;  const ll inf = 1e10;  const ll mod = 1e9+7;  ll n,q;  ll a[pp];  ll f[pp],g[pp];  map<ll,ll> m;  ll xxx[pp];  ll yyy[pp];  ll findFirst(ll x,ll l,ll r) {  ll mid = (l+r)>>1;  ll res = 1;  while(l <= r)  {  mid = (l + r)>>1;  if(f[mid] >= x)  {  res = mid;  r = mid - 1;  }  else l = mid + 1;  }  return res;  }  void buildyyy(ll k,ll l,ll r)  {  if(l == r)  {  yyy[k] = g[l];  return ;  }  ll mid = (l + r)>>1;  buildyyy(2\*k,l,mid);  buildyyy(2\*k+1,mid+1,r);  yyy[k] = max(yyy[2\*k],yyy[2\*k+1]);  }  ll getzzzz(ll k,ll l,ll r,ll L,ll R)  {  if(L > r || R < l)return 0;  if(l >= L && r <= R)return yyy[k];  ll mid = (l + r)>>1;  return max(getzzzz(2\*k,l,mid,L,R),getzzzz(2\*k+1,mid+1,r,L,R));  }  void buildxxx(ll k,ll l,ll r)  {  if(l == r)  {  xxx[k] = f[l];  return ;  }  ll mid = (l + r)>>1;  buildxxx(2\*k,l,mid);  buildxxx(2\*k+1,mid+1,r);  xxx[k] = max(xxx[2\*k],xxx[2\*k+1]);  }  ll gettttt(ll k,ll l,ll r,ll L,ll R)  {  if(L > r || R < l)return 0;  if(l >= L && r <= R)return xxx[k];  ll mid = (l + r)>>1;  return max(gettttt(2\*k,l,mid,L,R),gettttt(2\*k+1,mid+1,r,L,R));  }  int main()  {  ios\_base::sync\_with\_stdio(0);  cin>>n>>q;  FOR(i,1,n)  {  cin>>a[i];  f[i] = max(m[a[i]] + 1,f[i-1]);  g[i] = i - f[i] + 1;  m[a[i]] = i;  }  buildxxx(1,1,n);  buildyyy(1,1,n);  ll l,r;  while(q--)  {  cin>>l>>r;  l++;r++;  if(gettttt(1,1,n,l,r) <= l)  {  cout<<r-l + 1<<endl;  continue;  }  ll k = findFirst(l,l,r);  cout<<max(k-l,getzzzz(1,1,n,k,r))<<endl;  }  return 0;  } |

Mã nguồn gốc của mã nguồn trên như sau:

|  |
| --- |
| #include<bits/stdc++.h>  #define X first  #define Y second  typedef long long ll;  #define FOR(i,a,b) for(ll i = a;i<=b;i++)  #define FOD(i,a,b) for(ll i = a;i>=b;i--)  #define pb push\_back  #define mp make\_pair  using namespace std;  typedef pair<ll,ll>ii;  typedef priority\_queue<ll,vector<ll>,greater<ll> > heap\_min;  const ll maxN = 1e6+5;  const ll inf = 1e10;  const ll mod = 1e9+7;  ll n,q;  ll a[maxN];  ll f[maxN],g[maxN];  map<ll,ll> m;  ll tf[maxN];  void buildtf(ll k,ll l,ll r)  {  if(l == r)  {  tf[k] = f[l];  return ;  }  ll mid = (l + r)>>1;  buildtf(2\*k,l,mid);  buildtf(2\*k+1,mid+1,r);  tf[k] = max(tf[2\*k],tf[2\*k+1]);  }  ll tg[maxN];  void buildtg(ll k,ll l,ll r)  {  if(l == r)  {  tg[k] = g[l];  return ;  }  ll mid = (l + r)>>1;  buildtg(2\*k,l,mid);  buildtg(2\*k+1,mid+1,r);  tg[k] = max(tg[2\*k],tg[2\*k+1]);  }  ll getMaxF(ll k,ll l,ll r,ll L,ll R)  {  if(L > r || R < l)return 0;  if(l >= L && r <= R)return tf[k];  ll mid = (l + r)>>1;  return max(getMaxF(2\*k,l,mid,L,R),getMaxF(2\*k+1,mid+1,r,L,R));  }  ll getMaxG(ll k,ll l,ll r,ll L,ll R)  {  if(L > r || R < l)return 0;  if(l >= L && r <= R)return tg[k];  ll mid = (l + r)>>1;  return max(getMaxG(2\*k,l,mid,L,R),getMaxG(2\*k+1,mid+1,r,L,R));  }  ll findFirst(ll x,ll l,ll r)  {  ll mid = (l+r)>>1;  ll res = 1;  while(l <= r)  {  mid = (l + r)>>1;  if(f[mid] >= x)  {  res = mid;  r = mid - 1;  }  else l = mid + 1;  }  return res;  }  int main()  {  ios\_base::sync\_with\_stdio(0);  cin>>n>>q;  FOR(i,1,n)  {  cin>>a[i];  f[i] = max(m[a[i]] + 1,f[i-1]);  g[i] = i - f[i] + 1;  m[a[i]] = i;  }  buildtf(1,1,n);  buildtg(1,1,n);  ll l,r;  while(q--)  {  cin>>l>>r;  l++;r++;  if(getMaxF(1,1,n,l,r) <= l)  {  cout<<r-l + 1<<endl;  continue;  }  ll k = findFirst(l,l,r);  cout<<max(k-l,getMaxG(1,1,n,k,r))<<endl;  }  return 0;  } |