TÌM KIẾM NHỊ PHÂN

Tran Viet Khoa
tvkhoa.husc@gmail.com

Hue University— 01 September 2020

Mục lục

1 Giới thiệu

Bài viết dựa trên một loạt bài giảng về **Binary Search** của **ITMO Academy** với khóa học tên là **pilot**, các bạn có thể tham khảo bản gốc ở địa chỉ https://codeforces.com/edu/course/2/lesson/6. Dựa trên bài viết đó tôi viết lại bằng tiếng Việt và dùng Python để cài đặt, qua quá trình tìm hiểu tôi đưa ra một vài nhận xét để các bạn có thể hiểu rõ hơn về bài viết gốc.

Mỗi bài viết về chủ đề này (tổng số có năm bài về chủ đề này) sẽ bao gồm phần lý thuyết và phần thực hành. Bạn cần một handle trên codeforces để nộp bài, và có thể dùng nhiều ngôn ngữ lập trình khác nhau. Dùng C, C++ theo ý đồ của tác giả https://codeforces.com/profile/pashka hoặc bất kỳ ngôn ngữ yêu thích nào.

Cấu trúc của mỗi bài giảng gồm hai phần, một là một video trình bày lý thuyết và code mẫu được hướng dẫn chi tiết, hai là một practice với một số bài toán, đơn giản bạn chỉ cần học theo và nộp bài (submit). Tuy nhiên sẽ hay hơn nếu tự làm và tìm hiểu kỹ thuật toán. Do vậy, bài viết này cũng bao gồm hai phần như bài gốc.

2 Lý thuyết

Sau khi nghiên cứu kỹ lý thuyết và các bài tập của bài viết tôi nhận thấy đây là một bài khởi đầu dạng đơn giản của thuật toán tìm kiếm nhị phân, các phiên bản bổ sung của nó đều được hầu hết các ngôn ngữ lập trình cao cấp trang bị.

Đơn cử thư viện STL của C++ với các hàm `binary_search(), upper_bound, lower_bound()`, Python với các hàm thuộc thư viện `bisect` như `bisect.bisect_right(), bisect.bisect_left()`. Nghĩa là nếu bạn không cài đặt cũng như hiểu rõ chi tiết của từng thuật toán thì vẫn có thể vận dụng các hàm dựng sẵn (build-in) trên để nộp bài.

Phát biểu bài toán:

Cho dãy số $A = \{a_1, a_2, \dots a_n\}$ thỏa điều kiện $a_i \leq a_{i+1}$. Tìm phần tử x xem có xuất hiện trong dãy trên hay không, nghĩa là tồn tại i sao cho $a_i = x$. Nếu tìm thấy trả về vị trí tìm thấy $a_i = x$ (hoặc đơn giản trả về True), ngược lại trả về giá trị -1 (-1 là tự quy ước, hoặc False).

Xét ví dụ cho dãy $A = \{1, 2, 5, 6, 7, 9, 12, 14, 15, 32\}, x = 6$, kết quả tìm kiếm là $r = 3$. Xét $x = 3 -> r = -1$ hoặc trả về False.

Đây là một bài toán cực kỳ cơ bản cho các hệ thống ứng dụng trong tin học, xuất hiện hầu khắp mọi nơi được gọi là bài toán tìm kiếm. Dựa trên yêu cầu này mà các phương án tìm kiếm lời giải, cải tiến thuật toán để cho ra đời nhiều thuật toán tìm kiếm hơn, nhiều cấu trúc dữ liệu từ tuyến tính cho đến phi tuyến hỗ trợ cho việc tìm kiếm thông tin sao cho tốt nhất. Các thuật toán AI cũng tham gia mở rộng ra lĩnh vực nghiên cứu dựa trên bài toán cơ bản này.

Quay lại với bài toán. Ý tưởng đề xuất cho nó lần lượt như sau:

2.1 Thuật toán naive- Tìm kiếm tuần tự

Chính là thuật toán tìm kiếm tuần tự, ý tưởng lần lượt so sánh phần tử cần tìm x với các phần tử của dãy tính từ trái sang, nếu tìm thấy kết thúc tìm kiếm và trả về vị trí tìm thấy, nếu duyệt hết mà không tìm thấy thì gán kết quả là -1 (tự quy ước). Một thuật toán cực kỳ dễ cài đặt, mã giả như sau:

SequenceSearch

```
Tìm kiếm tuần tự
input: A, x
output: position of x=A[i] if find else -1
methode:
Begin
   i:=1;
   while (i<=n and x!=A[i]) do
      inc(i);
   if (i<= n) then
      return i (*return True*)
   else
      return -1; (*return False*)
End;
```

Độ phức tạp của thuật toán là $O(n)$. Nếu để chạy thuật toán này cho lập trình thi đấu thì với khoảng $n \leq 10^7 - 10^8$ chạy mất một giây, tham số chuẩn của máy chấm.

2.2 Thuật toán tìm kiếm nhị phân

Một điểm rất đặc biệt của bài toán mà thuật toán trên không khai thác được chính là $a_i \leq a_{i+1}$, nghĩa là dãy đã có thứ tự không giảm. Ý tưởng chia để trị (divede and conquer) đề xuất cho bài toán này gọi là tìm kiếm nhị phân (binary search).

Xét ví dụ về thuật toán tìm kiếm nhị phân như sau: Cuốn sách bạn đang đọc có 1000 trang và có đánh số trang, và ngày hôm qua bạn đã đọc đến trang thứ 347 mà không có đánh dấu nào trên sách cả. Bây giờ bạn cần tìm đến trang 347 để đọc tiếp và cách bạn tìm là như thế nào?

Cách 1. Bạn bắt đầu mở sách và bắt đầu từ trang 1 bạn lật liên tiếp cho đến trang thứ 347. Chính là thuật toán naive ở mục 2.1 trên. Sẽ mỏi tay nếu trang cần đọc là 997 nhỉ.

Cách 2. Bạn áng chừng (ước lượng) trang 347 và lật ở vị trí áng chừng đó và xem đó là trang số mấy. Nếu nó là số trang số 500, lớn hơn số 347 thì bạn có cần phải tìm trong nữa sau của quyển sách (từ trang 500 trở đi) nữa không?. Chắc chắn là không, như vậy mình đã loại đi gần nữa quyển sách cần tìm. Và bạn lặp lại hành động tương tự để tìm đến trang 347 với nửa đầu của quyển sách, cứ thế đến khi bạn tìm được trang số 347.

Do để biểu diễn sự áng chừng (ước lượng [1]) khi lật trang là khó cho nên để đơn giản người ta chọn vị trí ở chính giữa và được xác định một cách rất đơn giản bằng kỹ thuật hai con trỏ (two points), một trỏ bên trái (L) và một trỏ bên phải (R) quản lý toàn bộ số trang của quyển sách.

Thuật toán mô phỏng đệ quy như sau:

[1] Ý tưởng này chính là sự cải tiến của thuật toán tìm kiếm nhị phân. Gọi là chọn chốt theo xác suất

BinarySearch

```
Tìm kiếm nhị phân
input: A, x
output: position of x=A[i] if find else -1
methode recursive
function Bs(A, L, R, x) return Integer
Begin
   if (L > R) then return -1
   else
   begin
      mid:=(L+R)/2;
      if (A[mid]=x) then return mid
      else
         if (A[mid] < x) then return Bs(A, mid+1, R, x)
         else return Bs(A, L, mid-1, x);
   end;
End;
```

Độ phức tạp của thuật toán là O(logN) nhanh hơn rất nhiều so với thuật toán naive 2.1 ở trên. Việc chia dãy nhị phân trả về vị trí chính xác của phần tử cần tìm nếu dãy số là khóa (key) và có thứ tự không giảm.

Nếu dãy vẫn đảm bảo thứ tự không giảm nhưng có nhiều phần tử trùng nhau, vậy khi đó thuật toán 2.2 trên trả về vị trí của phần tử cần tìm là vị trí nào?. Xét ví dụ cho dãy $A = \{1, 2, 5, 6, 6, 6, 6, 7, 9, 12, 14, 15\}$, $x = 6$, kết quả tìm kiếm là $r = 6$. Tại sao không là 3 mà là 6, lời giải thích sẽ gặp ở phần thực hành 3.1

3 Thực hành

3.1 Problem A

Phát biểu bài toán: Ở đây tôi dịch lại nội dung bài toán, nếu bạn thích đọc nguyên bản tiếng Anh thì xem link ở 3.1.

Bài tập 1: Cài đặt thuật toán Binary Search

Cài đặt thuật toán binary_search và ứng dụng.

Input

- Dòng thứ nhất chứa hai số nguyên dương n, k thỏa $1 \leq n, k \leq 10^5$, n là kích thước của dãy và k câu truy vấn.
- Dòng thứ hai chứa n phần tử của dãy có thứ tự không giảm, mỗi phần tử có giá trị tuyệt đối nhỏ hơn hoặc bằng 10^9.
- Dòng thứ ba chứa k phần tử là các số nguyên cho các truy vấn.

Output

- Ứng với mỗi truy vấn in ra YES nếu tìm thấy và NO nếu ngược lại, mỗi truy vấn in trên một dòng.

Samples

Input samples
10 10
1 61 126 217 2876 6127 39162 98126 712687 1000000000
100 6127 1 61 200 -10000 1 217 10000 1000000000
Output samples
NO
YES
YES
YES
NO
NO
YES
YES
NO
YES

Nhận xét và giải:

Đây là bài toán đơn giản chỉ cài đặt lại nội dung của thuật toán tìm kiếm, tuy nhiên bạn không thể dùng thuật toán naive 2.1. Lý do độ phức tạp sẽ là $O(n \times k) = 10^{10}$ không đủ thời gian chạy mà phải cài đặt thuật toán binarySearch 2.2 khi đó độ phức tạp chỉ là $O(k \times LogN)$ đủ chạy với $1 \leq n, k \leq 10^5$.

Mã nguồn sau được viết bằng Python:

A.py

```python
#function bSearch: Thuật toán lặp
def bSearch(List, x):
   L, R  = 0, len(List)-1 #Chỉ mục python của List bắt đầu từ 1 đến n-1.
   while L <= R:
      mid = (L+R)// 2
      if List[mid] == x:
         return mid #Có thể return True.
      elif List[mid] < x:
         L = mid + 1
      else:
         R = mid - 1
   return -1 #Có thể return False.
#Hàm chính
n, k = map(int, input().split())
List = [int(i) for i in input().split()]
query = [int(i) for i in input().split()]
for i in query:
   if bSearch(List, i) != -1:
      print('YES')
   else:
      print('NO')
```

Nhận xét:

+ Bài này có thể dùng hàm `binary_search()` của STL C++ để nộp bài.

+ Nếu dãy số của mình có nhiều phần tử bằng nhau thì thuật toán có trả về vị trí xuất hiện đầu tiên tính từ trái sang như thuật toán tìm kiếm tuần tự ở trên, hoặc giả là trả về vị trí xuất hiện đầu tiên tính từ phải sang. Câu trả lời là có thể không trả về vị trí mong muốn như ví dụ $A = \{1, 2, 5, 6, 6, 6, 6, 7, 9, 12, 14, 15\}$, $x = 6$, kết quả tìm kiếm là 6 mà không phải 3.

+ Với nhận xét như trên, hai bài toán sau chính là phiên bản khác của thuật toán tìm kiếm nhị phân: Tìm vị trí chèn một phần tử vào danh sách có thứ tự sao cho sau khi chèn danh sách vẫn đảm bảo tính thứ tự, như vậy ở đây có hai vị trí chèn rất đặc biệt đó là vị trí bên trái nhất và bên phải nhất mà vẫn đảm bảo tính tăng dần.

+ Ví dụ xét dãy $A = \{1, 1, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 6\}$. Nếu phần tử cần chèn là 4 thì ở đây có hai vị trí đặc biệt. Vị trí thứ nhất tính từ trái qua đó là vị trí thứ 7, vị trí thứ hai tính từ phải sang là vị trí thứ 10.

Demo cho hai bài toán trên bằng hai hàm build-in của python

Command Line

```
>>> L = [1, 1, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 6]
>>> i = bisect.bisect_left(L, 4)
>>> j = bisect.bisect_right(L, 4)
>>> print (i, j)
```

Hai bài toán B và C sau chính là hai bài toán mà ta đề cập, tuy nhiên được viết theo cách nhìn khác của tác giả.

3.2 Problem B

Phát biểu bài toán: Ở đây tôi dịch lại nội dung bài toán, nếu bạn thích đọc nguyên bản tiếng Anh thì xem link ở 3.2.

Bài tập 2: Gần bên trái nhất

Cho một dãy số nguyên $A = \{a_1, a_2, \ldots, a_n\}$ sắp thứ tự không giảm và k truy vấn. Với mỗi truy vấn, tìm chỉ mục lớn nhất của một phần tử trong dãy không lớn hơn phần tử cho trước.

Input

- Dòng thứ nhất chứa hai số nguyên dương n, k thỏa $0 \leq n, k \leq 10^5$, n là kích thước của dãy và k câu truy vấn.
- Dòng thứ hai chứa n phần tử của dãy có thứ tự không giảm, mỗi phần tử có giá trị tuyệt đối nhỏ hơn hoặc bằng 2.10^9.
- Dòng thứ ba chứa k phần tử là các số nguyên cho các truy vấn.

Output

- Ứng với mỗi truy vấn in ra chỉ số lớn nhất của một phần tử mảng không lớn hơn số đã cho, nếu không có in 0. Mỗi số in trên một dòng.

Samples

Input samples
5 5
3 3 5 8 9
2 4 8 1 10
Output samples
0
2
4
0
5

Nhận xét:

+ Đây chính là thuật toán `bisect_right()` của thư viện bisect và hoàn toàn tương tự thuật toán `upper_bound()` của STL C++. Và chính là bài toán tìm vị trí chèn tính từ phải sang.

+ Cách cài đặt của tác giả gần như dự trên một trong những phiên bản cài đặt hàm `upper_bound()` của STL C++. Các bạn có thể tham khảo cách cài đặt bằng C++ và submit.

Ở phần cài đặt, thay vì dùng chỉ số $L = -1$ như tác giả, ở đây vẫn dùng được cho python tuy nhiên do python có sử dụng chỉ số -1 khi truy cập dãy cho nên khi viết tôi cảm giác bị ngượng. Bản cài đặt này tương tự như cách dùng hàm `bisect_right()`.

B.py

```
from sys import stdin
def bisect_right(a, x, lo=0, hi=None):
"""Return the index where to insert item x in list a, assuming a is sorted.
The return value i is such that all e in a[:i] have e < x, and all e in
a[i:] have e >= x.  So if x already appears in the list, a.insert(x) will
insert just before the leftmost x already there.
Optional args lo (default 0) and hi (default len(a)) bound the
slice of a to be searched.
"""
   if lo < 0:
      raise ValueError('lo must be non-negative')
   if hi is None:
      hi = len(a)
   while lo < hi:
      mid = (lo + hi) >> 1
      if x < a[mid]:  # Change <= to <, and you get bisect_right.#
         hi = mid
      else:
         lo = mid + 1
   return lo
"""Main function"""
d = [int(x) for x in stdin.readline().strip().split()]
L = [int(ele) for ele in stdin.readline().strip().split()]
query = [int(ele) for ele in stdin.readline().strip().split()]
for i in query:
   print(bisect_right(L, i))
```

Hoàn toàn tương tự vậy cho bài C.

3.3 Problem C

Phát biểu bài toán: Ở đây tôi dịch lại nội dung bài toán, nếu bạn thích đọc nguyên bản tiếng Anh thì xem link ở 3.3.

Bài tập 3: Gần bên trái nhất

Cho một dãy số nguyên $A = \{a_1, a_2, \ldots, a_n\}$ sắp thứ tự không giảm và k truy vấn. Với mỗi truy vấn, tìm chỉ mục nhỏ nhất của một phần tử trong dãy không nhỏ hơn phần tử cho trước.

Input

- Dòng thứ nhất chứa hai số nguyên dương n, k thỏa $0 \leq n, k \leq 10^5$, n là kích thước của dãy và k câu truy vấn.
- Dòng thứ hai chứa n phần tử của dãy có thứ tự không giảm, mỗi phần tử có giá trị tuyệt đối nhỏ hơn hoặc bằng 2.10^9.
- Dòng thứ ba chứa k phần tử là các số nguyên cho các truy vấn.

Output

- Ứng với mỗi truy vấn in ra chỉ số nhỏ nhất của một phần tử mảng không nhỏ hơn số đã cho, nếu không có in n+1. Mỗi số in trên một dòng.

Samples

Input samples
5 5
3 3 5 8 9
2 4 8 1 10
Output samples
1
3
4
1
6

Nhận xét và giải:

Warning: Đây chính là thuật toán `bisect_left()` của thư viện bisect và hoàn toàn tương tự thuật toán `lower_bound()` của STL C++.

C.py

```
from sys import stdin
def bisect_left(a, x, lo=0, hi=None):
"""Return the index where to insert item x in list a, assuming a is sorted.

The return value i is such that all e in a[:i] have e < x, and all e in
a[i:] have e >= x.  So if x already appears in the list, a.insert(x) will
insert just before the leftmost x already there.

Optional args lo (default 0) and hi (default len(a)) bound the
slice of a to be searched.
"""
   if lo < 0:
      raise ValueError('lo must be non-negative')
   if hi is None:
      hi = len(a)
   while lo < hi:
      mid = (lo + hi) >> 1
      if x <= a[mid]:  # Change <= to <, and you get bisect_right.#
         hi = mid
      else:
         lo = mid + 1
   return lo

""" main() function"""
d = [int(x) for x in stdin.readline().strip().split()]
L = [int(ele) for ele in stdin.readline().strip().split()]
query = [int(ele) for ele in stdin.readline().strip().split()]
for i in query:
   print(bisect_left(L, i)+1)
```

3.4 Problem D

Bài D chỉ là ứng dụng của hai hàm trên với ý tưởng sao chép một đoạn các phần tử bằng với x trong dãy. Mời các bạn đọc và submit bài.

D.py

```
n = int(input())
lst = sorted(list(map(int, input().split())))
k = int(input())
for _ in range(k):
l, r = list(map(int, input().split()))
print(bisect_right(lst, r) - bisect_left(lst, l), end=' ')
```