Các thuật toán tìm kiếm

1.1. Thuật toán tìm kiếm là gì ?
Thuật toán tìm kiếm được thiết kế để kiểm tra 1 phần tử hoặc truy xuất một phần tử từ bất kỳ cấu trúc dữ liệu nào mà nó được lưu trữ.
1.2. Phân loại.
Dựa vào loại tìm kiếm, các thuật toán tìm kiếm này thường được phân loại thành 2 loại chính:

Tìm kiếm tuần tự Trong tìm kiếm này, danh sách hoặc mảng được duyệt tuần tự và mỗi phần tử được kiểm tra. Ví dụ: Tìm kiếm tuần tự (Linear Search).

Tìm kiếm khoảng: Những thuật toán này được thiết kế đặc biệt để tìm kiếm trong cấu trúc dữ liệu đã được sắp xếp. Loại tìm kiếm này hiệu quả hơn nhiều so với Tìm kiếm tuần tự vì chúng nhắm đến trung tâm của cấu trúc tìm kiếm liên tục và chia không gian tìm kiếm làm hai nửa. Ví dụ: Tìm kiếm nhị phân (Binary Search).

2.1 Tìm kiếm tuần tự (Linear Search) là gì ?
Tìm kiếm tuần tự (Linear Search) được xác định là một thuật toán tìm kiếm tuần tự bắt đầu từ một đầu và duyệt qua từng phần tử của danh sách cho đến khi tìm thấy phần tử mong muốn, nếu không thì tìm kiếm tiếp tục cho đến cuối tập dữ liệu.
2.2 Tìm kiếm tuần tự hoạt động như thế nào ?
Thuật toán Tìm kiếm tuần tự hoạt động như thế nào?
Trong thuật toán Tìm kiếm tuần tự:

- Mỗi phần tử được xem xét như là một phần tử có thể khớp với khóa (key) và kiểm tra điều này.
- Nếu bất kỳ phần tử nào được tìm thấy bằng giá trị bằng với khóa, thì tìm kiếm thành công và trả về chỉ số của phần tử đó.
- Nếu không có phần tử nào được tìm thấy có giá trị bằng với khóa, thì tìm kiếm trả về kết quả "Không tìm thấy khớp".

Ví dụ: Xem xét mảng arr[] = {10, 50, 30, 70, 80, 20, 90, 40} và khóa (key) = 30.

- **Bước 1:** Bắt đầu từ phần tử đầu tiên (chỉ số 0) và so sánh khóa (key) với mỗi phần tử (arr[i]).

  So sánh khóa với phần tử đầu tiên arr[0]. Vì không bằng nhau, trình lặp di chuyển đến phần tử tiếp theo như một phần tử có thể khớp.

  So sánh khóa với phần tử tiếp theo arr[1]. Vì không bằng nhau, trình lặp di chuyển đến phần tử tiếp theo như một phần tử có thể khớp.

- **Bước 2:** Bây giờ, khi so sánh arr[2] với khóa (key), giá trị khớp. Vì vậy, thuật toán Tìm kiếm tuần tự sẽ trả về một thông báo thành công và trả về chỉ số của phần tử khi tìm thấy khóa (ở đây là 2).

2.3 Phân tích Độ phức tạp của Tìm kiếm tuần tự:

Độ phức tạp thời gian:

- Trường hợp tốt nhất: Trong trường hợp tốt nhất, khóa có thể có mặt tại chỉ số đầu tiên. Vì vậy, độ phức tạp tốt nhất là O(1).
- Trường hợp xấu nhất: Trong trường hợp xấu nhất, khóa có thể có mặt tại chỉ số cuối cùng, tức là đối diện với cuối từ nơi tìm kiếm đã bắt đầu trong danh sách. Vì vậy, độ phức tạp trong trường hợp xấu nhất là O(N), trong đó N là kích thước của danh sách.
- Trường hợp trung bình: O(N)

Không gian phụ: O(1) vì ngoài biến để lặp qua danh sách, không có biến nào được sử dụng.

Ưu và nhược điểm của Tìm kiếm tuần tự:

Ưu điểm:

- Tìm kiếm tuần tự có thể được sử dụng mà không cần xem xét danh sách có được sắp xếp hay không. Nó có thể được sử dụng trên mảng của bất kỳ loại dữ liệu nào.
- Không yêu cầu bộ nhớ bổ sung.
- Nó là một thuật toán phù hợp cho các tập dữ liệu nhỏ.

Nhước điểm:

- Tìm kiếm tuần tự có độ phức tạp thời gian là O(N), điều này làm cho nó chậm đối với các tập dữ liệu lớn.
- Không phù hợp cho các mảng lớn.

Khi nào sử dụng Tìm kiếm tuần tự?

- Khi chúng ta đang làm việc với một tập dữ liệu nhỏ.
- Khi bạn đang tìm kiếm một tập dữ liệu được lưu trữ trong bộ nhớ liền kề.0

2.4. Kỹ thuật lính canh

Có bao nhiều phép so sánh ?

Bài toán: Tìm kiếm phần tử x trên mảng một chiều có n phần tử.

Tiếp cận:

```
for(int i = 0;  i < n; i++)
if (arr[i] == x) return i;
//==============
i = 0;
while (i < n && arr[i] != x)
i++;
if (i < n) return i;
else return -1;
```

Đặt vấn đề: Tốn bao nhiều phép so sánh trong trường hợp xấu nhất ?

Ta có: Kỹ thuật lính canh

```
arr[n] = target;
int i = 0;
while (arr[i] != target)
    i++;
```

if (i < n) return i;
else return -1;

3.1 Binary Search là gì ?

Tìm kiếm nhị phân được xác định là một thuật toán tìm kiếm được sử dụng trong một mảng đã được sắp xếp bằng cách liên tục chia đôi khoảng tìm kiếm.

3.2 Binary Search hoạt động như thế nào ?

Trong thuật toán này,

- Chia không gian tìm kiếm thành hai phần bằng cách tìm chỉ số giữa "mid".
- Tìm chỉ số giữa "mid" trong thuật toán Tìm kiếm nhị phân.
- So sánh phần tử ở giữa của không gian tìm kiếm với khóa cần tìm.
- Nếu khóa được tìm thấy ở phần tử giữa, quá trình được kết thúc.
- Nếu khóa không được tìm thấy ở phần tử giữa, chọn nửa nào sẽ được sử dụng làm không gian tìm kiếm tiếp theo.
- Nếu khóa nhỏ hơn phần tử giữa, sử dụng phần bên trái cho tìm kiếm tiếp theo.
- Nếu khóa lớn hơn phần tử giữa, sử dụng phần bên phải cho tìm kiếm tiếp theo.
- Quá trình này tiếp tục cho đến khi khóa được tìm thấy hoặc không gian tìm kiếm tổng cộng được dùng hết.

Để hiểu cách hoạt động của tìm kiếm nhị phân, hãy xem minh hoạ sau đây:

Xem xét một mảng arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}, và mục tiêu (target) là 23.

Bước đầu tiên:
Tính toán chỉ số giữa (mid) và so sánh phần tử giữa với khóa cần tìm (key). Nếu khóa nhỏ hơn phần tử giữa (mid), di chuyển sang phía trái, và nếu khóa lớn hơn phần tử giữa, di chuyển không gian tìm kiếm sang phía phải.
Khóa (tức là 23) lớn hơn phần tử giữa hiện tại (tức là 16). Không gian tìm kiếm di chuyển sang bên phải.
Khóa nhỏ hơn phần tử giữa hiện tại là 56. Không gian tìm kiếm di chuyển sang bên trái.
Bước thứ hai: Nếu khóa khớp với giá trị của phần tử giữa, phần tử đó được tìm thấy và tìm kiếm dừng lại.

Làm thế nào để cài đặt Tìm kiếm nhị phân?
Thuật toán Tìm kiếm nhị phân có thể được thực hiện theo hai cách sau đây:

1. Thuật toán Tìm kiếm nhị phân lặp lại (Iterative Binary Search Algorithm)
2. Thuật toán Tìm kiếm nhị phân đệ quy (Recursive Binary Search Algorithm)

3.3 1. Thuật toán Tìm kiếm nhị phân lặp (Iterative Binary Search Algorithm):
Ở đây, chúng ta sử dụng một vòng lặp while để tiếp tục quá trình so sánh khóa và chia không gian tìm kiếm thành hai nửa.

int search(int arr[], int n, int x)

```
{
   int l = 0, r = n - 1;
   while (l <= r)
   {
      int mid = (l+r)/2;
      if (arr[mid] == x) return mid; // Nếu phần tử thứ arr thứ mid sẽ trả về giá trị mid
      if (arr[mid] < x) l = mid + 1; // nếu arr[mid] nhỏ hơn thì dịch chuyển qua phần bên phải đề
tìm kiếm
      else r = mid - 1; // ngược lại thì dịch chuyển qua phần bên trái để tìm kiếm
   }
   return -1;
}
```

Độ phức tạp: O(log N)
Không gian phụ: O(1)

3.3.2 Thuật toán Tìm kiếm nhị phân bằng đệ quy (Recursive Binary Search Algorithm)
Tạo một hàm đệ quy và so sánh giữa điểm giữa của không gian tìm kiếm với khóa (giá trị cần tìm). Dựa vào kết quả của phép so sánh, ta sẽ trả về chỉ số nơi tìm thấy khóa hoặc gọi lại hàm đệ quy để tiếp tục tìm kiếm trong không gian tìm kiếm tiếp theo.

```
int search_ver2(int arr[], int l, int r, int x)
{
   if (l <= r)
   {
      int mid = (l+r)/2;
      if (arr[mid] == x) return mid;
      if (arr[mid] < x) return search_ver2(arr,mid+1,r,x);
      else return search_ver2(arr, l, mid - 1, x);
   }
   else return -1;
}
```

Độ phức tạp: O(log N)
Không gian phụ: O(1), Nếu xem xét ngăn xếp cuộc gọi đệ quy thì không gian phụ sẽ là O(logN).
Ưu điểm của Tìm kiếm nhị phân:
1. Tìm kiếm nhị phân nhanh hơn tìm kiếm tuyến tính, đặc biệt là đối với mảng có kích thước lớn.
2. Hiệu quả hơn so với các thuật toán tìm kiếm khác có cùng độ phức tạp thời gian, như tìm kiếm nội suy hoặc tìm kiếm mũ.
3. Tìm kiếm nhị phân thích hợp cho việc tìm kiếm trong các tập dữ liệu lớn được lưu trữ trong bộ nhớ bên ngoài, chẳng hạn như trên ổ cứng hoặc trong đám mây.

Nhược điểm của Tìm kiếm nhị phân:
1. Mảng phải được sắp xếp.

2. Tìm kiếm nhị phân yêu cầu cấu trúc dữ liệu đang được tìm kiếm được lưu trữ trong các vị trí bộ nhớ liền kề.
3. Tìm kiếm nhị phân yêu cầu các phần tử trong mảng có thể so sánh được, nghĩa là chúng phải có thể được sắp xếp.

Ứng dụng của Tìm kiếm nhị phân:
1. Tìm kiếm nhị phân có thể được sử dụng như một khối xây dựng cho các thuật toán phức tạp hơn được sử dụng trong học máy, chẳng hạn như các thuật toán để huấn luyện mạng neural hoặc tìm kiếm siêu tham số tối ưu cho một mô hình.
2. Nó có thể được sử dụng để tìm kiếm trong đồ họa máy tính, chẳng hạn như các thuật toán cho việc theo dõi tia hoặc ánh xạ các vật liệu (texture mapping).
3. Nó có thể được sử dụng để tìm kiếm trong cơ sở dữ liệu.

Binary Search functions in C++ STL (binary_search, lower_bound and upper_bound)
1. binary_search:
binary_search(start_ptr, end_ptr, num): Hàm này trả về giá trị true nếu phần tử có mặt trong container, ngược lại trả về false. Biến start_ptr giữ địa chỉ bắt đầu của tìm kiếm nhị phân và biến end_ptr giữ vị trí kết thúc của không gian tìm kiếm nhị phân, còn num là giá trị cần tìm.
2. Lower_bound:
lower_bound(start_ptr, end_ptr, num): trả về con trỏ trỏ tới vị trí phần tử nhỏ nhất lớn hơn hoặc bằng nó lớn hơn hoặc bằng num.
3. Upper_bound:
upper_bound(start_ptr, end_ptr, num): trả về con trỏ trỏ tới vị trí phần tử nhỏ nhất lớn hơn num.
Code mẫu: sử dụng 3 function.
Thực hành
Linear_search:
https://cses.fi/problemset/task/1083

Binary_Search:

B - Binary Search - Codeforces (Chặt nhị phân đáp án số thực)
C - Binary Search - Codeforces (Chặt nhị phân đáp án số nguyên)

Bài 1:
Bạn được cung cấp tất cả các số từ 1 -> n ngoại trừ 1 số. Hãy tìm số bị thiếu đó.
Đầu vào:
Dòng đầu tiên: chứa 1 số nguyên n. (2 <= n <= 2.10^5)
Dòng thứ 2: chứa n - 1 số nguyên (mỗi số là riêng biệt và giá trị trong đoạn 1 -> n)
Đầu ra:
In ra số còn thiếu

Ví dụ:

input:
5
2 3 1 5
Output:
4

Ở đây, mình sẽ giới thiệu 2 cách đơn giản nhất để giải bài này.
Cách 1: dùng 1 mảng bool có kích thước n + 1 để đánh dấu, số nào có ở dòng số 2 thì chúng ta sẽ đánh dấu là true, và sau đó số bị thiếu là vị trí không được đánh dấu.

Cách này thì chúng ta sẽ có độ phức tạp là O(n), và bộ nhớ phụ lên đến O(n) vì chúng ta có tạo một mảng kích thước n + 1 để đánh dấu (số 1 không đáng kể)

Cách 2:
Như mọi người đã biết thì tổng các số tự nhiên từ 1 đến n chúng ta có thể tính được trong O(1) đúng không
Tong = n*(n+1)/2;
Vậy thì trong bài này tổng của n - 1 được cho có phải là tổng của n số tự nhiên từ 1 đến n trừ đi cho số bị thiếu đúng không ?
Vậy thì lật ngược lại vấn đề thì chúng ta sẽ tìm được số bị thiếu bằng cách lấy tổng của n số trừ đi cho tổng của (n-1) số được cho
Như vậy chúng ta chỉ tốn bộ nhớ phụ O(1) vì chỉ tạo 1 biến để lưu tổng của (n-1) số thôi.
Vậy thì chốt lại cách giải quyết của chúng ta sẽ là tính tổng của n số tự nhiên từ 1 đến n bằng công thức Tong ở trên và tính thêm tổng của của n - 1 số được cho và cuối cùng kết quả ta đem Tổng (n) số - Tổng (n-1) số thì sẽ ra được kết quả.