

Tìm kiếm nhị phân

Thuật toán tìm kiếm nhị phân

Nguồn: [Topcoder](#) 

Người dịch:

- Nguyễn Nhật Minh Khôi - VNU University of Science. Biên soạn lại từ bản dịch của Vũ Thị Thiên Anh

Reviewer: Hoàng Xuân Nhật, Trần Quang Lộc

Tìm kiếm nhị phân (hay còn gọi là chắt nhị phân) là một trong số các thuật toán cơ bản của khoa học máy tính. Trong bài viết này, chúng ta sẽ xây dựng một nền tảng lý thuyết, sau đó đưa ra cách cài đặt thuật toán này một cách chuẩn xác.

Bài toán mở đầu: Tìm giá trị cho trước trong một dãy đã sắp xếp

Mở đầu, ta sẽ đến với bài toán sử dụng tìm kiếm nhị phân đơn giản nhất. Đề bài như sau:



Cho trước một dãy A được sắp tăng dần, trả về vị trí của phần tử có giá trị x trong A . Lưu ý: để đơn giản, ta giả sử các phần tử trong mảng A có giá trị phân biệt.

Ví dụ

Đầu tiên, ta sẽ xét một ví dụ để thấy được tư tưởng của thuật toán.

Cho $A = [0, 5, 13, 19, 2, 41, 55, 68, 72, 81, 98]$ và $x = 55$, thuật toán sẽ diễn ra như hình dưới:

KHÔNG GIAN TÌM KIẾM											
	1	2	3	4	5	6	7	8	9	10	11
LƯỢT 1	0	5	13	19	22	41	55	68	72	81	98
						↑					
						Trung vị					

KHÔNG GIAN TÌM KIẾM											
	1	2	3	4	5	6	7	8	9	10	11
LƯỢT 2	0	5	13	19	22	41	55	68	72	81	98
									↑		
									Trung vị		

KHÔNG GIAN TÌM KIẾM											
	1	2	3	4	5	6	7	8	9	10	11
LƯỢT 3	0	5	13	19	22	41	55	68	72	81	98
							↑				
							Trung vị				

Ở lượt tìm đầu tiên, không gian tìm kiếm là tập hợp $S = \{1, \dots, 11\}$ gồm tất cả các chỉ số của mảng. Bắt đầu với việc chọn **phần tử trung vị** của không gian tìm kiếm hiện tại (chính là 6), ta nhận xét $A[6] = 41 < 55 = x$. Do theo đề bài mảng A được sắp xếp tăng dần, ta biết được tất cả các phần tử có chỉ số $1, \dots, 6$ đều nhỏ hơn giá trị cần tìm x . Do đó, chúng *chắc chắn không thể là kết quả*, khi đó không gian tìm kiếm có thể thu hẹp lại $S = \{7, \dots, 11\}$, tức *giảm đi một nửa*.

Tương tự, ở lượt tìm thứ hai, ta xét phần tử trung vị của không gian tìm kiếm hiện tại (chính là 9), nhận thấy $A[9] = 72 > 55 = x$. Cũng do mảng A được sắp xếp tăng dần, ta biết được tất cả các phần tử có vị trí từ $9, \dots, 11$ đều lớn hơn giá trị cần tìm x . Do đó, chúng *chắc chắn không thể là kết quả*, khi đó không gian tìm kiếm sẽ lại giảm đi một nửa $S = \{7, \dots, 8\}$.

Ở lượt tìm cuối cùng, ta cũng xét phần tử trung vị của không gian tìm kiếm hiện tại ở vị trí 7 (ở đây số lượng phần tử của không gian tìm kiếm là chẵn, do đó có hai phần tử trung vị, ta có thể chọn một trong hai đều được, ở ví dụ này ta chọn phần tử trung vị đầu tiên), Nhận thấy $A[7] = 55 = x$, ta kết luận 5 chính là vị trí của phần tử cần tìm và dừng thuật toán.

Tổng quát hóa bài toán

Từ ví dụ trên, ta có thể dễ dàng hiểu được ý tưởng của thuật toán tìm kiếm nhị phân. Đúng như tên gọi, thuật toán sẽ liên tục chia không gian tìm kiếm thành hai nửa và loại một nửa đi. Thuật toán có thể trình bày như sau:

1. Ta duy trì một không gian tìm kiếm S là một dãy con các giá trị có thể là kết quả (ở đây là chỉ số các phần tử trong A). Ban đầu, không gian tìm kiếm là toàn bộ các chỉ số của mảng $S = \{1, \dots, n\}$ với n là chỉ số phần tử cuối cùng của A .
2. Ở mỗi bước, thuật toán so sánh giá trị cần tìm với phần tử có chỉ số là trung vị trong không gian tìm kiếm. Dựa trên sự so sánh đó, cộng thêm việc ta biết dãy A có thứ tự, ta có thể loại một nửa số phần tử của S . Lặp đi lặp lại quá trình này, cuối cùng ta sẽ được một không gian tìm kiếm bao gồm một phần tử duy nhất.
3. Khi đó, nếu phần tử duy nhất đó bằng với giá trị cần tìm x thì đó là nghiệm của bài toán, nếu không thì bài toán vô nghiệm.

Ở đây có hai lưu ý khi cài đặt thuật toán:

- Do không gian tìm kiếm luôn là một đoạn liên tục các giá trị nguyên, ta không cần lưu tất cả phần tử của không gian khi tìm kiếm mà chỉ cần duy trì hai biến **low** và **high** tượng trưng cho phần tử đầu và cuối của đoạn.
- Ta có thể tối ưu thuật toán hơn bằng việc *dừng sớm* nếu trong quá trình so sánh gặp một phần tử trung vị thỏa yêu cầu đề bài chứ không cần đợi đến khi không gian tìm kiếm chỉ còn một phần tử.

Dưới đây là code minh họa viết bằng ngôn ngữ C++. Trong trường hợp giá trị cần tìm không tồn tại trong khoảng tìm kiếm thì thuật toán sẽ trả về -1

```

1  int binary_search(int A[], int sizeA, int target) {
2      int lo = 1, hi = sizeA;
3      while (lo <= hi) {
4          int mid = lo + (hi - lo)/2;
5          if (A[mid] == target)
6              return mid;
7          else if (A[mid] < target)
8              lo = mid+1;
9          else
10             hi = mid-1;
11     }
12     // không tìm thấy giá trị target trong mảng A
13     return -1;
14 }
```

Độ phức tạp thuật toán

Ở mỗi bước, kích thước không gian tìm kiếm bị giảm đi một nửa. Ta dễ thấy rằng độ phức tạp của thuật toán là $O(\log(N))$ với N là số phần tử ban đầu của không gian tìm kiếm.

Hàm \log là một hàm tăng rất chậm. Ví dụ như nếu phải tìm kiếm giá trị trong 1 triệu phần tử, với tìm kiếm nhị phân chỉ cần tối đa là 21 bước.

Tìm kiếm nhị phân trong thư viện chuẩn STL

C++ Standard Template Library đã cài đặt sẵn tìm kiếm nhị phân bằng các hàm [lower_bound](#), [upper_bound](#), [binary_search](#), [equal_range](#), bạn đọc có thể tham khảo tùy thuộc vào mục đích sử dụng.

Tìm kiếm nhị phân tổng quát

Ở phần trước, ta đã xét dạng đơn giản nhất của tìm kiếm nhị phân. Trong phần này, chúng ta sẽ tổng quát hóa thuật tìm kiếm nhị phân cho một lớp bài toán rộng hơn. Ta sẽ thấy tìm kiếm nhị phân có thể mở rộng để áp dụng cho **bất kỳ loại hàm số đơn điệu** nào nhận tham số đầu vào là số nguyên. Nói một cách đơn giản, một hàm số đơn điệu là *một hàm tăng hoặc một hàm giảm*. Trong ví dụ đầu bài, hiển nhiên mảng sắp xếp tăng có thể xem như một "hàm tăng".

Cơ sở lý thuyết: Định lý chính của tìm kiếm nhị phân

Khi gặp một bài toán mà ta đoán được có thể dùng tìm kiếm nhị phân để giải, thì ta phải chứng minh tính đúng đắn suy luận của chúng ta. Do đó, xây dựng một cơ sở lý thuyết vững chắc là vô cùng cần thiết. Sau đây, tôi sẽ

trình bày một lớp tổng quát hóa nữa cho các bài toán có thể áp dụng tìm kiếm nhị phân, song song đó là ví dụ thực tế với bài toán mở đầu.

Cho không gian tìm kiếm S bao gồm các ứng cử viên cho kết quả của bài toán. Ta định nghĩa một hàm kiểm tra $P : S \rightarrow \{\text{true}, \text{false}\}$ là hàm nhận một ứng cử viên $x \in S$ và trả về giá trị **true**/**false** cho biết x có hợp lệ hay không (tùy vào bài toán mà định nghĩa hợp lệ sẽ khác nhau). Hiểu đơn giản, hàm P là hàm "kiểm tra" một tính chất nào đó, xem một ứng cử viên cho kết quả của bài toán có thỏa tính chất đó không.

Với ví dụ ở đầu bài, thay vì tìm chỉ số của phần tử có giá trị 55, ta có thể viết lại đề bài thành "tìm chỉ số nhỏ nhất sao cho phần tử ở chỉ số đó lớn hơn hoặc bằng 55". Khi đó, không gian tìm kiếm ban đầu $S = \{1, \dots, 11\}$ (ban đầu mọi chỉ số của mảng đều có thể là kết quả) và $P(x) = \text{boolean}(a[x] \geq 55)$ trả về **true** nếu $a[x] \geq 55$ và **false** nếu $a[x] < 55$.

Định lý chính (Main Theorem) cho biết rằng: một bài toán chỉ có thể áp dụng tìm kiếm nhị phân nếu và chỉ nếu hàm kiểm tra P của bài toán thỏa mãn

$$\forall x, y \in S, y > x \wedge P(x) = \text{true} \Rightarrow P(y) = \text{true} \quad (*)$$

Lưu ý rằng tính chất trên của hàm kiểm tra P cũng tương đương với tính chất sau:

$$\forall x, y \in S, y < x \wedge P(x) = \text{false} \Rightarrow P(y) = \text{false} \quad (**)$$

Sự tương đương ở đây có thể chứng minh bằng [phương pháp phản chứng](#) \square , để tránh bài viết quá dài dòng, phần chứng minh để lại cho bạn đọc.

Định lý chính mang cho chúng ta một thông tin rất quan trọng, đó là **điều kiện cần và đủ để một bài toán có thể giải bằng tìm kiếm nhị phân**. Để hiểu được tại sao, chúng ta hãy phân tích kỹ hơn ý nghĩa tính chất của hàm P mà định lý yêu cầu

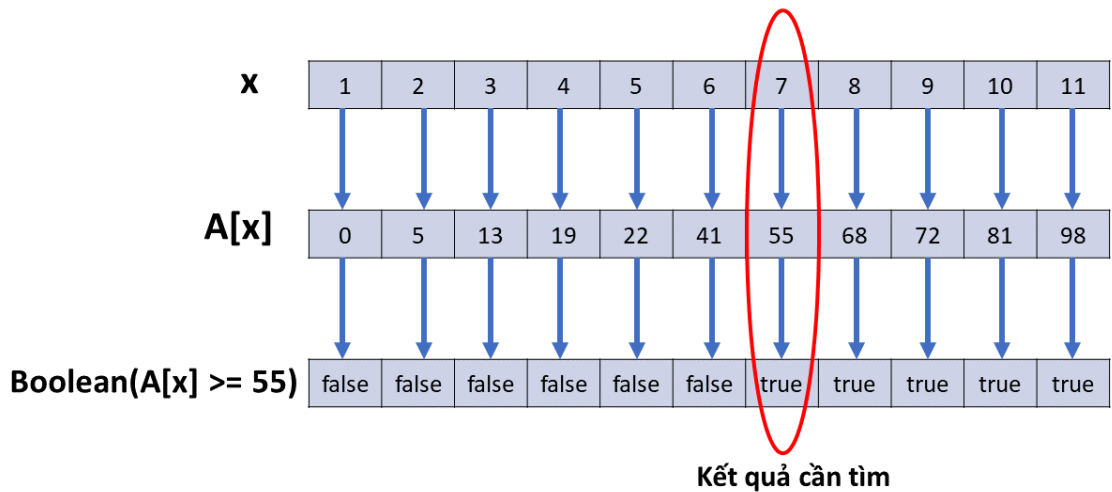
- Tính chất **(*)** có thể giải thích như sau: **nếu x hợp lệ thì mọi phần tử $y > x$ đều hợp lệ**. Tính chất này giúp chúng ta loại đi nửa sau của không gian tìm kiếm do đã biết chắc x là phần tử nhỏ nhất trong nửa sau hợp lệ, ta ghi nhận x là kết quả tạm thời và tiếp tục tìm xem có phần tử nào ở nửa đầu (nhỏ hơn x) hợp lệ hay không.
- Tương tự, tính chất **(**)** có thể giải thích như sau: **nếu x không hợp lệ thì mọi phần tử $y < x$ đều không hợp lệ**. Tính chất này giúp chúng ta loại đi nửa trước của không gian tìm kiếm do đã biết chắc chúng không hợp lệ, ta chỉ quan tâm những phần tử ở nửa sau (lớn hơn x) mà ta chưa biết thông tin chúng có hợp lệ hay không.

Nếu ta tính giá trị $P(x)$ cho từng phần tử trong S ban đầu, ta sẽ được một dãy liên tiếp các giá trị **false** liên tiếp rồi một dãy liên tiếp các giá trị **true** (từ nay gọi là dãy $P(S)$). Dễ thấy ta có thể áp dụng tìm kiếm nhị phân trên dãy $P(S)$ mới này để tìm giá trị x *nhỏ nhất* thỏa mãn $P(x) = \text{true}$ (hoặc cũng có thể làm cách tìm giá trị x *lớn nhất* mà $P(x) = \text{false}$, tuy nhiên ở đây ta không chọn cách này).

Với ví dụ đầu bài, như đã nói $P(x) = \text{boolean}(A[x] \geq 55)$. Dễ thấy P thỏa mãn tính chất đầu tiên, do A được sắp tăng dần nên nếu $A[x] \geq 55$ thì chắc chắn các phần tử y sau x đều thỏa $A[y] \geq A[x] \geq 55$. Tương tự ta cũng suy ra được, nếu $A[x] < 55$ thì chắc chắn các phần tử y trước x đều thỏa $A[y] \leq A[x] < 55$.

Áp dụng hàm $P(x) = \text{boolean}(A[x] \geq 55)$ cho từng phần tử của $S = \{1, \dots, 11\}$ ta có hình

sau



Chú ý rằng ta thấy có thể dễ dàng xây dựng định lý chính dựa trên một hàm kiểm tra P ngược lại, tức $P(S)$ sẽ là một dãy **true** liên tiếp theo sau bởi **false** liên tiếp. Tuy nhiên, ở đây ta sẽ chỉ xét một trường hợp để bài viết ngắn gọn hơn, trường hợp còn lại có thể làm tương tự.

Từ định lý trên, ta rút ra được mấu chốt để giải một bài toán dùng tìm kiếm nhị phân là ta cần **thiết kế được hàm P hợp lý sao cho thỏa mãn điều kiện trong định lý chính**.

Cuối cùng, tại sao ta phải tốn công tổng quát hóa thuật toán này thay vì dùng cách làm đơn giản ở ví dụ đầu? Đó là vì nhiều bài toán không thể ở dưới dạng tìm kiếm một giá trị cụ thể, nhưng ta lại có thể định nghĩa một hàm kiểm tra thỏa yêu cầu định lý chính để có thể áp dụng tìm kiếm nhị phân. Bằng cách đó, ta có thể mở rộng lớp bài toán có thể giải bằng tìm kiếm nhị phân.

Ví dụ điển hình cho việc áp dụng định lý là với bài toán *Tìm căn bậc hai*, thay vì hỏi "Số x nào bình phương lên thì bằng a ?" và tìm kiếm tuần tự tất cả các trường hợp, ta có thể định nghĩa hàm $P(x)$ trả lời cho câu hỏi " x^2 có lớn hơn hoặc bằng a hay không?" sau đó dùng tìm kiếm nhị phân để tìm x nhỏ nhất thỏa mãn. Với cách làm này ta có thể đơn giản hóa bài toán thành một bài toán yes/no, hơn thế còn giảm độ phức tạp của bài toán từ $O(n)$ xuống chỉ còn $O(\log(n))$.

Cài đặt thuật toán tổng quát

Trước khi cài đặt thuật toán, ta phải trả lời những câu hỏi sau:

1. Dãy $P(S)$ của bạn có dạng **false** — **true** (**false** liên tiếp rồi **true** liên tiếp) hay **true** — **false** (ngược lại)? Ở cài đặt phía dưới sẽ **mặc định dãy $P(S)$ có dạng false — true**, vì vậy nếu dãy $P(S)$ của bạn có dạng **true** — **false**, hãy đảo hàm $P(x)$ của bạn ngược lại.
2. Bài của bạn có luôn có nghiệm không? Nếu không hãy kiểm tra trước khi tìm kiếm nhị phân để tiết kiệm chi phí tính toán.
3. Mục tiêu của bạn là tìm phần tử **false** lớn nhất hay tìm phần tử **true** nhỏ nhất? Ở đây sẽ trình bày cả hai cách.

4. Nếu bài toán có nghiệm, hãy đảm bảo giá trị chặn dưới và chặn trên của khoảng tìm kiếm (biến `lo` và `hi`) là bắt đầu và kết thúc của một *khoảng đóng mà chắc chắn chứa kết quả cần tìm* (phần tử x đầu tiên mà $P(x) = \text{true}$). Hãy đảm bảo điều kiện này trong lúc thu hẹp không gian tìm kiếm để tránh xảy ra lỗi.
5. Phạm vi tìm kiếm đã đủ rộng chưa? Sẽ có nhiều lúc chấm xong bạn nhận ra là mình thiếu vài trường hợp biên. Vì thời gian chạy chỉ tăng theo hàm logarit $O(\log(N))$, bạn hoàn toàn có thể nâng rộng khoảng tìm kiếm ra mà ít khi lo bị quá thời gian. Tuy nhiên, phải để ý các lỗi như tràn mảng, tràn số,...
6. Luôn kiểm tra trường hợp $P(S) = [\text{false}, \text{true}]$. Để hiểu lý do tại sao hãy đọc *Trường hợp 2* của cài đặt.

TH1: tìm x nhỏ nhất mà $P(x) = \text{true}$. Dưới đây là đoạn code mẫu viết bằng C++.

```

1 | bool P(int x) {
2 |     // Logic của hàm P ở đây
3 |     return true; // thay giá trị này bằng giá trị đúng logic.
4 | }
5 |
6 | int binary_search(int lo, int hi) {
7 |     while (lo < hi) {
8 |         int mid = lo + (hi-lo)/2;
9 |         if (P(mid) == true)
10 |             hi = mid;
11 |         else
12 |             lo = mid+1;
13 |     }
14 |
15 |     if (P(lo) == false)
16 |         return -1; // P(x) = false với mọi x thuộc S, bài toán vô nghiệm.
17 |
18 |     return lo; // lo là giá trị x nhỏ nhất mà P(x) = true
19 | }
```

Hai dòng quan trọng là $hi = mid$ và $lo = mid + 1$, chúng giúp ta thu hẹp không gian tìm kiếm dần.

Khi $P(mid) = \text{true}$, ta có thể bỏ nửa sau của không gian tìm kiếm vì đã biết phần tử trong đó luôn hợp lệ. Tuy nhiên ta vẫn phải giữ mid trong không gian tìm kiếm mới vì nó có thể là phần tử đầu tiên mà $P = \text{true}$. Do đó không gian tìm kiếm mới sẽ là $S = \{lo, mid\}$, ta gán $hi = mid$.

Tương tự, khi $P(mid) = \text{false}$, ta có thể bỏ nửa đầu (bao gồm cả phần tử mid) vì tất cả các phần tử này đều không hợp lệ. Lúc này không gian tìm kiếm mới sẽ là $S = \{mid + 1, hi\}$, ta gán $lo = mid + 1$.

TH2: tìm x lớn nhất mà $P(x) = \text{false}$, suy luận tương tự như trên, ta có đoạn code như sau:

```

1 | bool P(int x) {
2 |     // Logic của hàm P ở đây
3 |     return true; // thay giá trị này bằng giá trị đúng logic.
4 | }
5 |
6 | int binary_search(int lo, int hi) {
7 |     while (lo < hi) {
```

```

8      int mid = lo + (hi-lo+1)/2;
9      if (P(mid) == true)
10         hi = mid - 1;
11     else
12         lo = mid;
13 }
14
15 if (P(lo) == true)
16     return -1; // P(x) = true với mọi x thuộc S, bài toán vô nghiệm.
17
18 return lo; // lo là giá trị x lớn nhất mà P(x) = false
19 }
```

Bạn sẽ thắc mắc rằng tại sao cách tính mid lại có một tí khác biệt so với thuật toán đầu tiên. Để hiểu được tại sao ta phải làm thế, ta sẽ xét trường hợp sau: trong quá trình tìm kiếm, nếu tại một thời điểm nào đó mà dãy $P(S)$ tạo ra bởi các phần tử của không gian tìm kiếm có dạng như sau

false	true
-------	------

Nếu ta tính $mid = lo + (hi - lo)/2$, đoạn code sẽ lặp vô hạn. Nó sẽ luôn chọn phần tử trung vị là $mid = lo$, nhưng cận dưới lo sẽ không di chuyển vì nó muốn giữ lại phần tử có $p = \text{false}$ thỏa yêu cầu tìm kiếm đó. Do đó, ta thay đổi công thức tính mid thành $mid = lo + (hi - lo + 1)/2$, làm như vậy sẽ khiến cận dưới sẽ được làm tròn lên thay vì làm tròn xuống, khi đó nó có thể loại bỏ phần tử **true** trước khi xét phần tử **false**. Có nhiều cách làm khác để thực hiện điều này, tuy nhiên, đây là cách dễ hiểu nhất. Do vậy, hãy luôn luôn kiểm tra thử trường hợp $P(S) = [\text{false}, \text{true}]$.

Một điều có thể bạn đang thắc mắc nữa là tại sao để tìm trung vị ta tính $mid = lo + (hi - lo)/2$ chứ không phải $mid = (hi + lo)/2$. Sở dĩ phải làm như vậy là để tránh khả năng xảy ra lỗi làm tròn số nguyên, ta muốn phép chia được làm tròn xuống, về gần với cận dưới, tuy nhiên phép chia làm tròn khác khi có số âm, nên nếu $(lo + hi)$ là số âm thì kết quả sẽ bị làm tròn lên. Code như trong mẫu kia giúp quá trình tính toán đều được làm tròn đúng theo logic. Đối với các bài toán mà chỉ cần xử lý giá trị dương thì lỗi này không xảy ra.

Cài đặt thuật toán với nửa khoảng

Cài đặt với đoạn đóng $[lo, hi]$ như trên có ưu điểm là dễ hiểu. Tuy nhiên, quay lại một chút với cơ sở lý thuyết: với dãy $P(S)$ có dạng **false** – **true**, thực tế ta nên chọn giá trị lo và hi mà $P(lo) = \text{false}$ và $P(hi) = \text{true}$ để đảm bảo luôn tìm được nghiệm. Do đó sẽ không ổn nếu như dãy $P(S)$ của ta đều toàn giá trị **false** (tức vô nghiệm). Trong trường hợp này, cài đặt với đoạn đóng có thêm đoạn kiểm tra để **return - 1**.

Giải pháp cho trường hợp này chính là **cài đặt với nửa khoảng**. Để ý rằng trong trường hợp vô nghiệm, cách cài đặt với khoảng đóng sẽ trả về cận trên hi của đoạn tìm kiếm ban đầu (do chỉ có cận dưới lo liên tục dịch chuyển lên và chạm tới hi). Hơn nữa, thuật toán của ta không bao giờ gọi $P(hi)$ vì để có $mid = hi$, ta đã phải có $lo = hi$, nhưng lúc đó thuật toán chắc chắn đã dừng do điều kiện **while (lo < hi)** trong cài đặt.

Điều này cho ta một ý tưởng: tạo một giá trị $hi' = hi + 1$ ảo và mặc định coi $P(hi') = \text{true}$. Ta sẽ tìm kiếm nhị phân trên đoạn $[lo, hi']$ mới và nếu bài toán vô nghiệm thì kết quả trả về sẽ là hi' , ta **không cần phải kiểm tra để return -1**. Sở dĩ gọi là cài đặt với nửa khoảng vì khi truyền tham số cho hàm là lo và hi , thực ra ta chỉ muốn kết quả trong nửa khoảng $[lo, hi)$, còn hi chỉ là giá trị ảo cho biết trường hợp vô nghiệm.

Cài đặt cũng tương tự với đoạn đóng, ngoại trừ việc ta loại bỏ đi đoạn code kiểm tra vô nghiệm:

```

1  bool P(int x) {
2      // Logic của hàm P ở đây
3      return true; // thay giá trị này bằng giá trị đúng logic.
4  }
5
6  // nhớ rằng ta phải truyền hi lớn hơn một đơn vị
7  // so với đoạn tìm kiếm thực sự
8  int binary_search(int lo, int hi) {
9      while (lo < hi) {
10         int mid = lo + (hi-lo)/2;
11         if (P(mid) == true)
12             hi = mid;
13         else
14             lo = mid+1;
15     }
16
17     return lo; // lo là giá trị x nhỏ nhất mà P(x) = true
18 }

```

Cách cài đặt này còn có một ưu điểm khác, đó là trong C++ và rất nhiều ngôn ngữ lập trình khác thì mảng bắt đầu từ 0, vì vậy nếu cần tìm một phần tử nào đó trong mảng thì với cài đặt bằng nửa khoảng tham số truyền vào sẽ rất đẹp, đó là `binary_search(0, N)` với N là số phần tử của mảng. Toàn bộ thư viện STL, `lower_bound`, `upper_bound` đều nhận nửa khoảng, và thực tế nguyên lý của các hàm đó cũng như trên: không tìm ra đáp án thì trả về `iterator end`.

Lưu ý: Với trường hợp ta muốn tìm vị trí phần tử `false` cuối cùng thì nửa khoảng cần tìm sẽ là `(lo, hi]` và hàm sẽ tự động trả về `lo` nếu mọi giá trị trong khoảng là `true`.

Ví dụ

Đến đây ta sẽ áp dụng những điều vừa học vào một bài cụ thể [Leetcode 1011](#) .

Trong bài này, một băng chuyền phải vận chuyển các gói hàng theo thứ tự cho trước trong $days$ ngày. Gói hàng thứ i có trọng lượng $weights[i]$. Biết mỗi ngày băng chuyền chỉ có thể vận chuyển tổng khối lượng tối đa là MAX . Đề bài yêu cầu tìm MAX nhỏ nhất để băng chuyền có thể hoàn thành nhiệm vụ được giao.

Để ý thấy rằng, với một số MAX , ta có thể tính toán được số ngày để chuyển hết các gói hàng sao cho mỗi ngày tổng khối lượng vận chuyển không quá MAX . Ban đầu, giao gói hàng 1 để ngày 1 chuyển. Nếu vận chuyển gói hàng 2 trong ngày 1 không làm tổng khối lượng trong ngày vượt MAX , thì ta sẽ chuyển luôn gói đó trong ngày 1. Nếu không, ta sẽ chuyển gói đó trong ngày 2. Làm lần lượt như thế tới khi mọi gói hàng đều được chuyển, ta sẽ có được số ngày tối thiểu cần để chuyển hết gói hàng với tổng khối lượng mỗi ngày không quá MAX .

Quay lại đề bài, ta thấy rằng thực ra vấn đề của bài toán là tìm số MAX nhỏ nhất sao cho số ngày tối thiểu để chuyển hết gói hàng không quá $days$ ngày. Như vậy, ta có thể dùng tìm kiếm nhị phân với hàm kiểm tra P được xây dựng bằng thuật toán tham lam được trình bày ở trên để kiểm tra các giá trị MAX . Để ý tính chất đơn

điều ở đây thể hiện ở chỗ nếu MAX tăng lên thì số lượng ngày cần dùng *chỉ có thể giữ nguyên hoặc giảm đi* nên hàm P thỏa định lý chính và có thể áp dụng tìm kiếm nhị phân.

Sau đây là đoạn code mẫu bằng C++:

```

1 // hàm kiểm tra P
2 bool check(int capacity, const vector<int>& weights, int days) {
3     int current_weight = 0; --days;
4     for(int i = 0; i < weights.size(); ++i) {
5         if (current_weight + weights[i] <= capacity)
6             current_weight += weights[i];
7         else {
8             --days;
9             current_weight = weights[i];
10        }
11    }
12    return days >= 0;
13 }
14
15 // hàm tìm kiếm nhị phân
16 int shipWithinDays(const vector<int>& weights, int days) {
17     int lo = 0, hi = 0;
18     for (int i = 0; i < weights.size(); ++i) {
19         lo = max(lo, weights[i]);
20         hi += weights[i];
21     }
22
23     while (lo < hi) {
24         int mid = lo + (hi - lo)/2;
25         if (check(mid, weights, days))
26             hi = mid;
27         else
28             lo = mid + 1;
29     }
30
31     return lo;
32 }

```

Hàm tìm kiếm nhị phân chính là hàm `shipWithinDays` và hàm kiểm tra là hàm `check`.

Có một lưu ý về việc chọn cận dưới và cận trên. Cận trên có thể là bất cứ số nguyên nào đủ lớn, ở đây chọn là tổng của tất cả các gói hàng (cho trường hợp tệ nhất cần vận chuyển trong đúng một ngày). Tuy nhiên cận dưới phải bằng ít nhất là khối lượng của gói hàng nặng nhất để tránh trường hợp gói hàng quá lớn để chuyển trong một ngày.

Để kiểm tra thuật toán không bị lặp vô hạn với trường hợp `[false, true]`, ta thử một test như sau: $weights = [1, 1]$, $days = 1$ và thấy thuật toán hoạt động tốt.

Độ phức tạp thuật toán là $O(n \cdot \log(SIZE))$ với $SIZE = hi - lo + 1$ là kích thước của không gian tìm kiếm và n là số lượng gói hàng, do đó thuật toán chạy rất nhanh.

Tìm kiếm nhị phân trên số thực

Tìm kiếm nhị phân cũng có thể được áp dụng khi không gian tìm kiếm là một đoạn số thực. Thường thì việc xử lý sẽ đơn giản hơn với số nguyên do ta không phải bận tâm về việc dịch chuyển cận:

```

1  bool P(double x) {
2      // Logic của hàm P ở đây
3      return true; // thay giá trị này bằng giá trị đúng logic.
4  }
5
6  bool isTerminated(double lo, double hi) {
7      // trả về kết quả của việc kiểm tra
8      // lo và hi có thỏa điều kiện dừng chưa
9  }
10
11 double binary_search(double lo, double hi) {
12     while (isTerminated(lo, hi) == false) {
13         double mid = lo + (hi-lo)/2;
14         if (P(mid) == true)
15             hi = mid;
16         else
17             lo = mid;
18     }
19     // trung bình cộng lo và hi xấp xỉ
20     // ranh giới giữa false và true
21     return lo + (hi-lo)/2;
22 }



```

Ta thường không tìm được giá trị mục tiêu một cách chính xác mà chỉ có thể xấp xỉ kết quả, đó là lý do có hàm điều kiện dừng `isTerminated`. Thông thường có 2 cách quyết định khi nào dừng vòng lặp:




1. **Dừng sau một số vòng lặp cố định (fixed):** thông thường khi làm bài tập trên TopCoder, chỉ cần lặp khoảng 100 lần là đủ (nhiều khi là thừa) để đạt được độ chính xác mong muốn cho những bài dạng thế này.
2. **Sai số tuyệt đối (absolute error):** dừng khi $hi - lo \leq \epsilon$ (ϵ thường rất nhỏ, khoảng 10^{-8}). Cách này được sử dụng nếu thời gian chặt và bạn phải tiết kiệm số lần lặp.



Một số bài toán về tìm kiếm nhị phân

Đơn giản

- [Power](#) 
- [Sushi for Two](#) 

Nâng cao

- [ACOW - USACO21 Open Silver](#) 
- [Wooden Sticks](#) 
- [c11cave](#) 

- [Increase and Copy](#) 
- [Prime Matrix](#) 

Được cung cấp bởi [Wiki.js](#)