**CÂY PHÂN ĐOẠN (SEGMENT TREE)**

## Phần I: LÝ THUYẾT

1. **Giới thiệu**

Cây phân đoạn là cấu trúc dữ liệu lưu trữ thông tin về các khoảng mảng dưới dạng cây. Nó cho phép trả lời các truy vấn trên mảng một cách hiệu quả, trong khi vẫn đủ linh hoạt để cho phép sửa đổi mảng nhanh chóng. Điều này bao gồm việc tìm tổng các phần tử mảng liên tiếp a[l..r] hoặc tìm phần tử tối thiểu trong một phạm vi như vậy với O(logn) thời gian. Giữa việc trả lời các truy vấn như vậy, cây phân đoạn cho phép sửa đổi mảng bằng cách thay thế một phần tử hoặc thậm chí thay đổi các phần tử của toàn bộ phân đoạn con (ví dụ gán tất cả các phần tử a[l..r] vào bất kỳ giá trị nào hoặc thêm giá trị cho tất cả phần tử trong phân đoạn). Nói chung, cây phân đoạn là một cấu trúc dữ liệu rất linh hoạt và có thể giải quyết được rất nhiều bài toán. Ngoài ra, cũng có thể áp dụng các thao tác phức tạp hơn và trả lời các truy vấn phức tạp hơn. Đặc biệt, cây phân đoạn có thể dễ dàng khái quát hóa thành các kích thước lớn hơn. Ví dụ, với cây phân đoạn hai chiều, ta có thể trả lời các truy vấn tổng hoặc tối thiểu trên một số hình chữ nhật con của ma trận đã cho chỉ trong O(log2n) thời gian. Một đặc tính quan trọng của cây phân đoạn là chúng chỉ yêu cầu một lượng bộ nhớ tuyến tính. Cây phân đoạn tiêu chuẩn yêu cầu 4n các đỉnh để làm việc trên một mảng kích thước n.

## Dạng đơn giản nhất của cây phân đoạn

Chúng ta xem xét dạng đơn giản nhất của cây phân đoạn khi muốn trả lời các truy vấn tổng một cách hiệu quả. Định nghĩa: Cho một mảng a[0...n-1], cây phân đoạn phải có khả năng tìm tổng các phần tử giữa các chỉ số l và r (tức là tính tổng a[l] + a[l+1] + … + a[r]), đồng thời xử lý việc thay đổi giá trị của các phần tử trong mảng (tức là thực hiện các phép gán có dạng a[i] = x). Cây phân đoạn sẽ có thể xử lý cả hai truy vấn trong O(logn) thời gian.

Việc triển khai mảng đơn giản - chỉ sử dụng một mảng - có thể cập nhật các phần tử trong O(1), nhưng đòi hỏi O(n) để tính toán từng truy vấn tổng. Và các tổng tiền tố được tính toán trước có thể tính toán các truy vấn tổng trong O(1) bằng prefix sum, nhưng việc cập nhật một phần tử mảng đòi hỏi O(n) thay đổi các tổng tiền tố.

## Cấu trúc của cây phân đoạn

Chúng ta có thể áp dụng cách tiếp cận chia để trị khi nói đến các phân đoạn mảng. Ta tính và lưu trữ tổng các phần tử của cả mảng, tức là tổng của phân đoạna[0...n-1]. Sau đó chia mảng thành hai nửa a[0...n/2-1] và a[n/2...n-1] và tính

tổng của mỗi nửa và lưu trữ chúng. Mỗi nửa trong số hai nửa này lần lượt được chia làm đôi và cứ tiếp tục làm như thế cho đến khi tất cả các phân đoạn đều đạt kích thước 1.

Có thể xem các đoạn này như một cây nhị phân: gốc của cây này là đoạn a[0...n-1] và mỗi đỉnh (trừ đỉnh lá) có đúng hai đỉnh con. Đây là lý do tại sao cấu trúc dữ liệu được gọi là "Cây phân đoạn".

Sau đây là minh họa của cây phân đoạn trên mảng a = [ 1, 3, -2, 8, -7]:

a[0…4]

sum = 3

a[0…2]

sum = 2

a[3…4]

sum = 1

a[0…1]

sum = 4

a[2…2]

sum = -2

a[3…3]

sum = 8

a[4…4]

sum = -7

a[0…0]

sum = 1

a[1…1]

sum = 3

Từ mô tả về cấu trúc dữ liệu trên, chúng ta có thể kết luận rằng cây phân đoạn chỉ yêu cầu số đỉnh tuyến tính. Cấp độ đầu tiên của cây chứa một nút duy nhất (gốc), cấp độ thứ hai sẽ chứa hai đỉnh, ở cấp độ thứ ba sẽ chứa bốn đỉnh, cho đến khi số đỉnh đạt tới n. Do đó, số đỉnh trong trường hợp xấu nhất có thể được ước tính bằng tổng 1 + 2 + 4 + … + 2[log2n] < 2[log2n] + 1 < 4n.

Chiều cao của cây phân đoạn là O(log n) vì khi đi từ gốc xuống lá kích thước các đoạn giảm đi khoảng một nửa.

## Xây dựng cây phân đoạn

Trước khi xây dựng cây phân đoạn, chúng ta cần quyết định:

1. ***Giá trị*** được lưu trữ tại mỗi nút của cây phân đoạn. Ví dụ, trong cây phân đoạn tổng, một nút sẽ lưu trữ tổng các phần tử trong phạm vi của nó [l, r]***.***
2. Thao tác ***hợp nhất*** để hợp nhất tất cả các nút trong một cây phân đoạn. Ví dụ trong cây phân đoạn tổng, hai nút tương ứng với phạm vi a[l1, r1] và a[l2, r2] sẽ được

hợp nhất thành một nút tương ứng với phạm vi a[l1, r2] bằng cách cộng các giá trị của hai nút.

Lưu ý rằng một đỉnh là "đỉnh lá", nếu đoạn tương ứng của nó chỉ bao gồm một giá trị trong mảng ban đầu. Nó đại diện ở cấp độ thấp nhất của cây phân đoạn. Giá trị của nó sẽ bằng phần tử tương ứng a[i].

Bây giờ, để xây dựng cây phân đoạn, chúng ta bắt đầu ở cấp độ dưới cùng (các đỉnh lá) và gán cho chúng các giá trị tương ứng. Trên cơ sở các giá trị này, chúng ta có thể tính toán các giá trị của cấp độ trước đó bằng cách sử dụng hàm “**hợp nhất”**. Và trên cơ sở đó, chúng ta có thể tính các giá trị trước đó và lặp lại quy trình cho đến khi đạt đến đỉnh gốc.

Sẽ thuận tiện hơn khi mô tả thao tác này một cách đệ quy theo hướng khác, tức là từ đỉnh gốc đến các đỉnh lá. Quy trình xây dựng, nếu được gọi trên một đỉnh không có lá, sẽ thực hiện như sau:

1. *Xây dựng đệ quy các giá trị của hai đỉnh con*
2. *Hợp nhất các giá trị tính toán của những đỉnh con này.*

Chúng ta bắt đầu xây dựng ở đỉnh gốc và do đó, chúng ta có thể tính toán toàn bộ cây phân đoạn. Độ phức tạp về thời gian của việc xây dựng này là O(n), giả sử rằng thao tác hợp nhất có thời gian không đổi (thao tác hợp nhất được gọi n lần, bằng số lượng nút bên trong trong cây phân đoạn).

## Truy vấn tổng

Bây giờ chúng ta sẽ trả lời các truy vấn về tổng. Mỗi một đầu vào, chúng ta nhận được hai số nguyên l và r; và chúng ta phải tính tổng của đoạn a[l … r] trong O(log n) thời gian. Để làm điều này, chúng ta sẽ duyệt qua cây phân đoạn và sử dụng tổng được tính toán trước của các phân đoạn. Giả sử rằng chúng ta hiện đang ở đỉnh bao phủ đoạn a[tl … tr]. Có ba trường hợp có thể xảy ra:

1. Trường hợp dễ nhất là khi đoạn a[l … r] bằng với đoạn tương ứng của đỉnh hiện tại (tức là a[l … r] = a[tl … tr]), thì chúng ta đã hoàn thành và có thể trả về tổng được tính toán trước được lưu trữ ở đỉnh.
2. Trường hợp thứ hai, phân đoạn truy vấn có thể rơi hoàn toàn vào miền của con bên trái hoặc con bên phải. Phần con bên trái bao phủ đoạn a[tl … tm] và đỉnh bên phải bao phủ đoạn a[tm+1 … tr] với tm = (tl + tr)/2. Trong trường hợp này,

chúng ta có thể chỉ cần đi đến đỉnh con, đoạn tương ứng bao gồm đoạn truy vấn và thực hiện thuật toán được mô tả với đỉnh đó.

1. Trường hợp cuối cùng, đoạn truy vấn giao nhau với cả hai phần tử con. Trong trường hợp này, chúng ta không có lựa chọn nào khác là thực hiện hai lệnh gọi đệ quy, một lệnh gọi cho mỗi phần tử con. Đầu tiên ta đi đến con bên trái, tính kết quả từng phần cho đỉnh này (tức là tổng các giá trị giao giữa đoạn truy vấn và đoạn con bên trái), sau đó sang con bên phải, tính kết quả từng phần sử dụng đỉnh đó, sau đó kết hợp các câu trả lời bằng cách cộng chúng lại. Nói cách khác, vì con bên trái đại diện cho phân đoạn a[tl … tm] và phân đoạn con bên phải là a[tm+1 … tr], chúng ta tính toán truy vấn tổng a[tl … tm] sử dụng con bên trái và truy vấn tổnga[tm+1 … tr**]** sử dụng con bên phải.

Vì vậy, việc xử lý một truy vấn tổng là một hàm gọi đệ quy chính nó một lần với con trái hoặc con phải (không thay đổi ranh giới truy vấn) hoặc hai lần, một lần cho con trái và một lần cho con bên phải (bằng cách chia truy vấn thành hai truy vấn con). Và quá trình đệ quy kết thúc, bất cứ khi nào ranh giới của phân đoạn truy vấn hiện tại trùng với ranh giới của phân đoạn của đỉnh hiện tại. Trong trường hợp đó, câu trả lời sẽ là giá trị được tính toán trước của tổng của phân đoạn này đã được lưu trữ trong cây.

Nói cách khác, việc tính toán truy vấn là một phép duyệt cây, qua tất cả các nhánh cần thiết của cây và sử dụng các giá trị tổng được tính toán trước của các phân đoạn trong cây.

Tại sao độ phức tạp của thuật toán này O(logn)? Để thể hiện sự phức tạp này, chúng ta xem xét từng cấp độ của cây. Đối với mỗi cấp độ, chúng ta chỉ truy cập không quá bốn đỉnh. Và vì chiều cao của cây là O(logn), chúng ta nhận được thời gian chạy mong muốn.

Chúng ta có thể chỉ ra rằng mệnh đề này (nhiều nhất là bốn đỉnh mỗi cấp) là đúng bằng quy nạp. Ở cấp độ đầu tiên, chúng ta chỉ thăm một đỉnh, đỉnh gốc, vì vậy ở đây chúng ta thăm ít hơn bốn đỉnh. Bây giờ chúng ta hãy nhìn vào một mức độ tùy ý. Theo giả thuyết quy nạp, ta đi thăm nhiều nhất 4 đỉnh. Nếu chúng ta chỉ truy cập tối đa hai đỉnh thì cấp độ tiếp theo tối đa bốn đỉnh. Điều đó không quan trọng vì mỗi đỉnh chỉ có thể thực hiện tối đa hai lệnh gọi đệ quy. Vì vậy, giả sử rằng chúng ta đi qua ba hoặc bốn đỉnh ở cấp độ hiện tại. Từ các đỉnh đó, chúng ta sẽ phân tích các đỉnh ở giữa kỹ hơn. Vì truy vấn tổng yêu cầu tổng của một mảng con liên tục nên chúng ta biết rằng các phân đoạn tương ứng với các đỉnh đã truy cập ở giữa sẽ được bao phủ hoàn toàn bởi phân đoạn của truy vấn tổng. Do đó, các đỉnh này sẽ không thực hiện bất kỳ lệnh gọi đệ quy nào. Vì vậy, chỉ có đỉnh bên trái nhất và đỉnh bên phải nhất mới có khả năng thực hiện các lệnh gọi đệ quy. Và những điều đó sẽ chỉ tạo ra tối đa bốn lệnh gọi đệ quy, do đó cấp độ tiếp theo cũng sẽ đáp ứng khẳng định. Chúng ta có thể nói rằng một nhánh tiếp cận ranh giới bên trái của truy vấn và nhánh thứ hai tiếp cận ranh giới bên phải.

Vì vậy chúng ta ghé thăm nhiều nhất 4logn tổng số đỉnh và bằng thời gian chạy của O(log n).

Tóm lại, truy vấn hoạt động bằng cách chia phân đoạn đầu vào thành nhiều phân đoạn con mà tất cả các tổng đã được tính toán trước và lưu trữ trong cây. Và nếu chúng ta ngừng phân vùng bất cứ khi nào đoạn truy vấn trùng với đoạn đỉnh thì chúng ta chỉ cần O(log n) các phân đoạn như vậy.

## Cập nhật truy vấn

Bây giờ chúng ta muốn sửa đổi một phần tử cụ thể trong mảng, giả sử chúng ta muốn thực hiện nhiệm vụ a[i] = x. Và chúng ta phải xây dựng lại cây phân đoạn sao cho nó tương ứng với mảng mới đã được sửa đổi.

Truy vấn này dễ hơn truy vấn tổng. Mỗi cấp độ của cây phân đoạn tạo thành một phân vùng của mảng. Do đó một phần tử a[i] chỉ đóng góp vào một phân khúc từ mỗi cấp độ. Như vậy chỉ O(logn) các đỉnh cần được cập nhật.

Dễ dàng nhận thấy rằng yêu cầu cập nhật có thể được thực hiện bằng hàm đệ quy. Hàm được truyền qua đỉnh cây hiện tại và nó gọi đệ quy chính nó bằng một trong hai đỉnh con (đỉnh chứa a[i] trong phân đoạn của nó) và sau đó tính toán lại giá trị tổng, tương tự như cách thực hiện trong phương thức xây dựng (tức là tổng của hai phần tử con của nó).

Ví dụ ở đây chúng ta thực hiện cập nhật **a[2] = 3**. Các đỉnh màu xanh lá cây là các đỉnh mà chúng ta truy cập và cập nhật.

a[0…4]

sum = 8

a[0…2]

sum = 7

a[3…4]

sum = 1

a[0…1]

sum = 4

a[2…2]

sum = 3

a[3…3]

sum = 8

a[4…4]

sum = -7

a[0…0]

sum = 1

a[1…1]

sum = 3

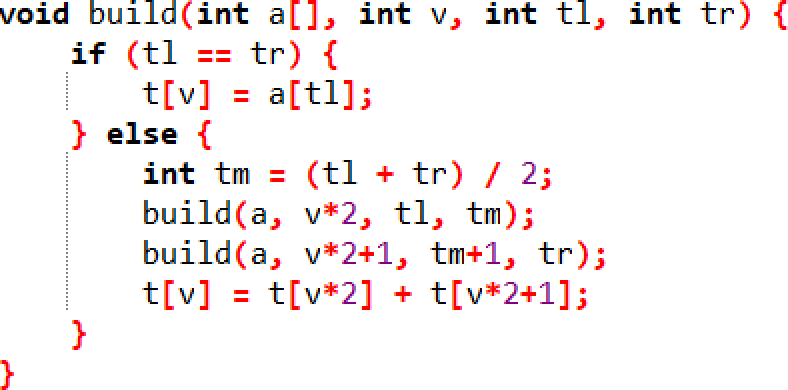
## Cài đặt:

Vấn đề cần quan tâm là cách lưu trữ cây phân đoạn. Có thể định nghĩa một cấu trúc **Vertex *(đỉnh)*** và tạo các đối tượng lưu trữ các ranh giới của phân đoạn, tổng của nó và ngoài ra còn trỏ đến các đỉnh con của nó. Tuy nhiên, điều này đòi hỏi phải lưu trữ rất nhiều thông tin dư thừa dưới dạng con trỏ. Chúng ta sẽ sử dụng một thủ thuật đơn giản để làm cho việc này hiệu quả hơn nhiều bằng cách sử dụng *cấu trúc dữ liệu ngầm* : Chỉ lưu trữ tổng trong một mảng (*một phương pháp tương tự được sử dụng cho cây nhị phân đầy đủ*). Tổng của đỉnh gốc tại chỉ số 1, tổng của hai đỉnh con của nó tại

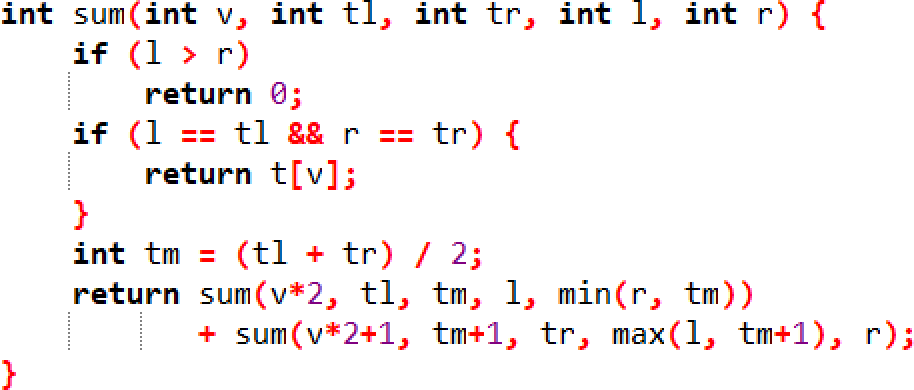
chỉ số 2 và 3, tổng các đỉnh con của hai đỉnh đó tại chỉ số 4 đến 7, v.v. Với chỉ mục 1, dễ thấy là con trái của đỉnh tại chỉ mục i được lưu trữ tại chỉ mục 2i và bên phải ở chỉ mục 2i + 1. Tương tự, cha của một đỉnh tại chỉ số i được lưu trữ tại i/2 (chia số nguyên).

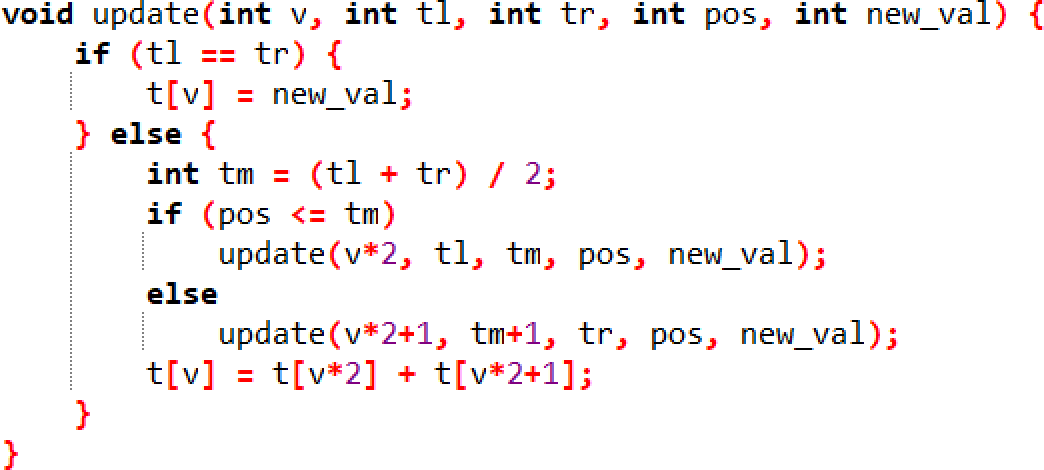
Điều này đơn giản hóa việc cài đặt rất nhiều. Chúng ta không cần lưu trữ cấu trúc của cây trong bộ nhớ do được định nghĩa ngầm. Chúng ta chỉ cần một mảng chứa tổng của tất cả các phân đoạn. Như đã lưu ý trước đó, chúng ta cần lưu trữ nhiều nhất 4n đỉnh. Nó có thể ít hơn, nhưng để thuận tiện, chúng ta luôn phân bổ một mảng có kích thước 4n. Sẽ có một số phần tử trong mảng tổng không tương ứng với bất kỳ đỉnh nào trong cây thực tế, nhưng điều này không làm phức tạp việc triển khai. Vì vậy, chúng ta lưu trữ cây phân đoạn đơn giản dưới dạng một mảng t[] với kích thước gấp bốn lần kích thước đầu vào n.

Quy trình cài đặt cây phân đoạn từ một mảng nhất định a[]: đó là hàm đệ quy với các tham số a[] (mảng đầu vào), v (chỉ số của đỉnh hiện tại) và các ranh giới tl và tr của phân đoạn hiện tại. Trong chương trình chính, hàm này sẽ được gọi với các tham số của đỉnh gốc: v = 1, tl = 0, và tr = n-1



Hơn nữa, hàm trả lời các truy vấn tổng cũng là hàm đệ quy, hàm này nhận thông tin tham số về đỉnh/đoạn hiện tại (tức là chỉ mục v và ranh giới tl và tr) và cả thông tin về ranh giới của truy vấn l và r. Để đơn giản hóa mã, hàm này luôn thực hiện hai lệnh gọi đệ quy, ngay cả khi chỉ cần một lệnh gọi - trong trường hợp đó, lệnh gọi đệ quy không cần thiết sẽ có l > r và điều này có thể dễ dàng bị phát hiện bằng cách sử dụng một kiểm tra bổ sung ở đầu hàm.



Cuối cùng là truy vấn cập nhật. Hàm cũng sẽ nhận thông tin về đỉnh/đoạn hiện tại và thêm vào đó là tham số của truy vấn cập nhật (tức là vị trí của phần tử và giá trị mới của nó).

## Cài đặt hiệu quả bộ nhớ

Hầu hết mọi người sử dụng cách triển khai từ phần trước. Nếu nhìn vào mảng t ta có thể thấy rằng nó tuân theo việc đánh số các nút cây theo kiểu BFS (duyệt cây theo chiều rộng). Sử dụng phép duyệt này cho các con của đỉnh v là 2v và 2v+1 tương ứng. Tuy nhiên, nếu n không phải là lũy thừa của 2, phương thức này sẽ bỏ qua một số chỉ số và để lại một số phần của mảng t không được sử dụng. Bộ nhớ bị giới hạn bởi 4n, mặc dù cây phân đoạn gồm một mảng n phần tử chỉ yêu cầu 2n -1 đỉnh.

Có thể được giảm bớt bằng cách đánh số lại các đỉnh của cây theo thứ tự của một đường đi Euler (*kiểu duyệt tiền thứ tự*) và viết tất cả các đỉnh này cạnh nhau. Chúng ta hãy nhìn vào một đỉnh tại chỉ mục v và nó quản lý phân đoạn[l, r] với mid = (l+r)/2. Vì vậy con bên trái sẽ có chỉ số v+1. Rõ ràng con bên trái chịu trách nhiệm về

phân đoạn [l … mid], tức là tổng cộng sẽ có 2 \* (mid - l + 1) - 1 các đỉnh của cây con bên trái. Vì vậy chúng ta có thể tính chỉ số của con phải của v. Chỉ số sẽ là v + 2 \* (mid - l + 1) - 1. Bằng cách đánh số này, chúng ta đạt được mức giảm bộ nhớ cần thiết để 2n.

## Một số phiên bản mở rộng của cây phân đoạn

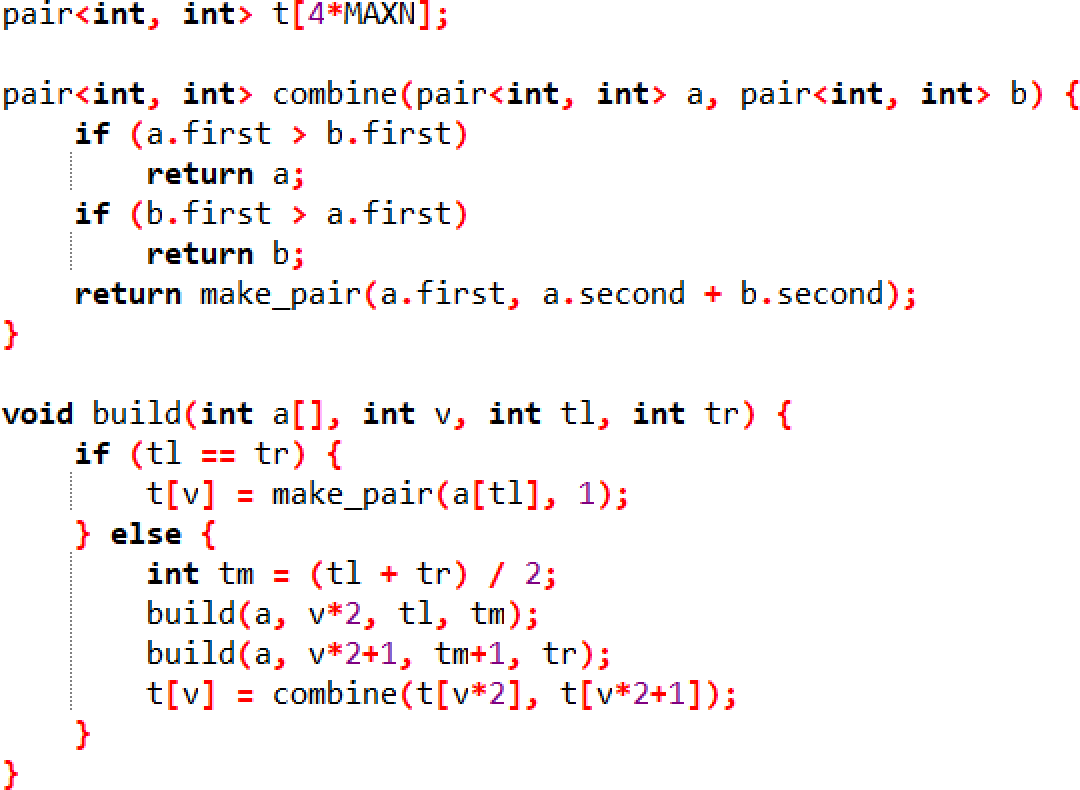
Cây phân đoạn là một cấu trúc dữ liệu linh hoạt và cho phép các biến thể và tiện ích mở rộng theo nhiều hướng khác nhau như sau:

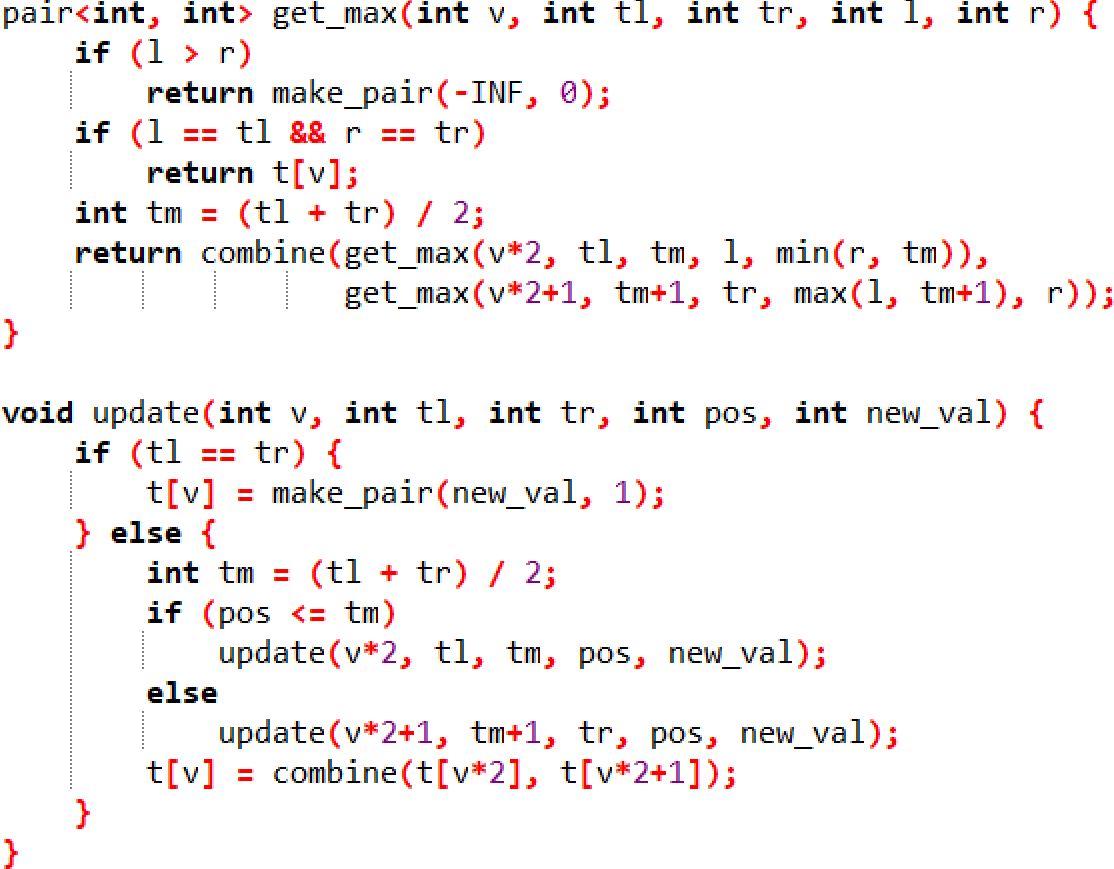
### Tìm giá trị lớn nhất

Chúng ta thay đổi một chút điều kiện của bài toán được mô tả ở trên: thay vì truy vấn tổng, bây giờ chúng ta sẽ thực hiện các truy vấn lớn nhất. Cây cũng sẽ có cấu trúc giống như cây được mô tả ở trên. Chúng ta chỉ cần thay đổi cách t[v] được tính toán trong hàm **built** và **update**. Bây giờ t[v] sẽ lưu trữ giá trị lớn nhất phân đoạn tương ứng. Và chúng ta cũng cần thay đổi cách tính giá trị trả về của hàm **sum** (thay thế tổng bằng lớn nhất). Tất nhiên bài toán này có thể dễ dàng chuyển thành phép tính giá trị nhỏ nhất thay vì giá trị lớn nhất.

### Tìm giá trị lớn nhất và số lần xuất hiện

Ngoài việc tìm giá trị lớn nhất, chúng ta còn phải tìm số lần xuất hiện của giá trị đó. Để giải quyết vấn đề này, chúng ta lưu trữ một cặp số ở mỗi đỉnh trong cây: Ngoài giá trị lớn nhất chúng ta còn lưu trữ số lần xuất hiện của nó trong đoạn tương ứng. Xác định chính xác cặp để lưu trữ tại t[v] có thể vẫn được thực hiện trong thời gian không đổi, bằng cách sử dụng các cặp thông tin được lưu trữ tại các đỉnh con. Việc kết hợp hai cặp như vậy nên được thực hiện trong một hàm riêng biệt, vì đây sẽ là thao tác mà chúng ta sẽ thực hiện khi xây dựng cây, đồng thời trả lời các truy vấn lớn nhất và khi thực hiện sửa đổi.





### Tính ước chung lớn nhất, bội chung nhỏ nhất

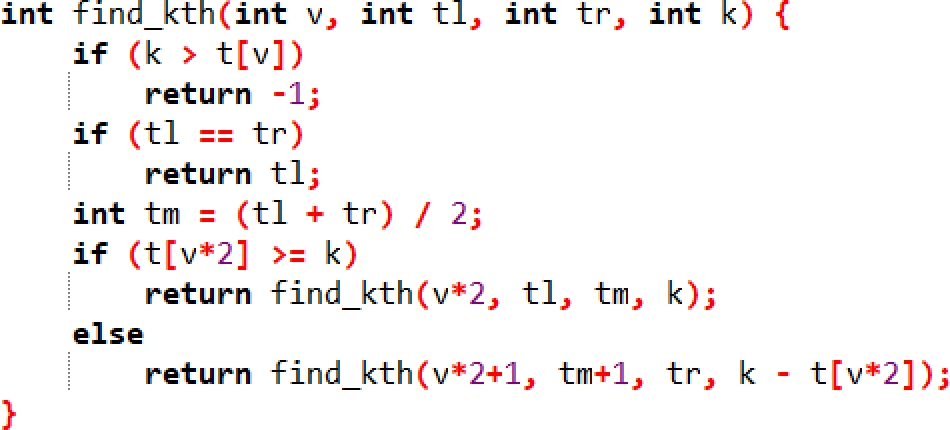
Trong bài toán này, chúng ta muốn tính GCD/LCM của tất cả các số trong phạm vi đã cho của mảng. Biến thể này của cây phân đoạn có thể được giải theo cách giống như cây phân đoạn mà chúng ta rút ra cho các truy vấn tổng/nhỏ nhất/lớn nhất:

chỉ cần lưu trữ GCD/LCM của đỉnh tương ứng trong mỗi đỉnh của cây là đủ. Việc kết hợp hai đỉnh có thể được thực hiện bằng cách tính GCD/LCM của cả hai đỉnh.

### Đếm số 0 và tím số 0 thứ k

Trong bài toán này, chúng ta muốn tìm số lượng số 0 trong một phạm vi nhất định và tìm thêm chỉ số của 0 thứ k bằng cách sử dụng hàm thứ hai. Một lần nữa chúng ta phải thay đổi giá trị lưu trữ của cây một chút: Lần này chúng ta sẽ lưu trữ số 0 trong mỗi đoạn trong t[]. Rõ ràng về cách thực hiện các hàm **built**, **update** và **count\_zero**, chúng ta có thể sử dụng các ý tưởng từ bài toán truy vấn tổng. Như vậy chúng ta đã giải quyết được phần đầu của vấn đề.

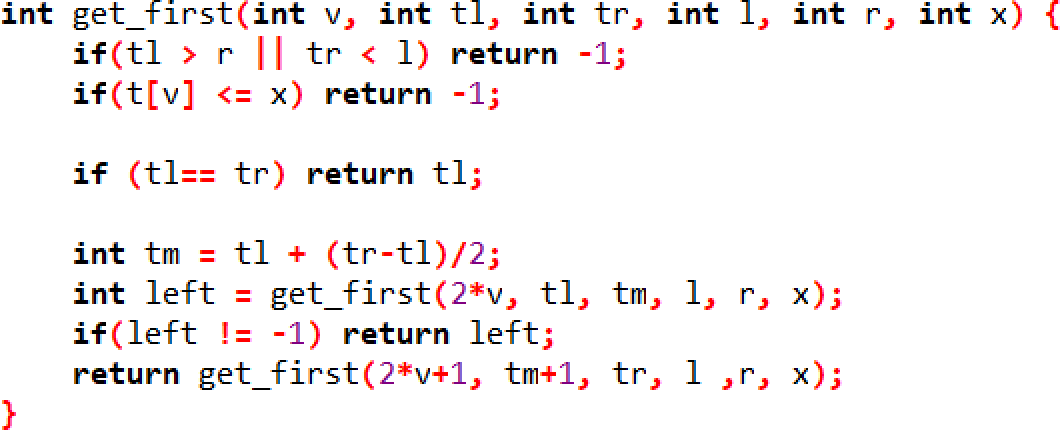
Bây giờ chúng ta tìm hiểu cách giải bài toán tìm số 0 thứ k trong mảng a[]. Để thực hiện việc này, chúng ta sẽ duyệt cây phân đoạn, bắt đầu từ đỉnh gốc và mỗi lần di chuyển sang con trái hoặc con phải, tùy thuộc vào đoạn nào chứa số 0 thứ k. Để quyết định xem chúng ta cần đi đến con nào, chỉ cần nhìn vào số lượng số 0 xuất hiện trên đoạn tương ứng với đỉnh bên trái là đủ. Nếu đếm trước lớn hơn hoặc bằng k cần phải chuyển xuống con bên trái, nếu không thì chuyển xuống con bên phải. Lưu ý, nếu chọn con bên phải, chúng ta phải trừ số 0 của con bên trái khỏi k. Trong quá trình thực hiện có thể xử lý trường hợp đặc biệt, a[] chứa ít hơn k số 0 thì trả về -1.



### Tìm kiếm mảng tiền tố với giá trị nhất định

*Bài toán như sau: Cho* giá trị x, yêu cầu phải nhanh chóng tìm chỉ số i nhỏ nhất sao cho tổng của i số đầu tiên của các phần tử của mảng a[] lớn hơn hoặc bằng x (giả sử rằng mảng a[] chỉ chứa các giá trị không âm).

Bài toán này có thể được giải quyết bằng cách sử dụng tìm kiếm nhị phân, tính tổng các tiền tố bằng cây phân đoạn. Tuy nhiên điều này sẽ dẫn tới độ phức tạp O(log2n). Thay vào đó, chúng ta có thể sử dụng ý tưởng tương tự như trong phần trước

và tìm vị trí bằng cách giảm dần cây: di chuyển sang trái hoặc sang phải mỗi lần, tùy thuộc vào tổng của con trái. Vì vậy việc tìm ra câu trả lời trong O(log n) thời gian.

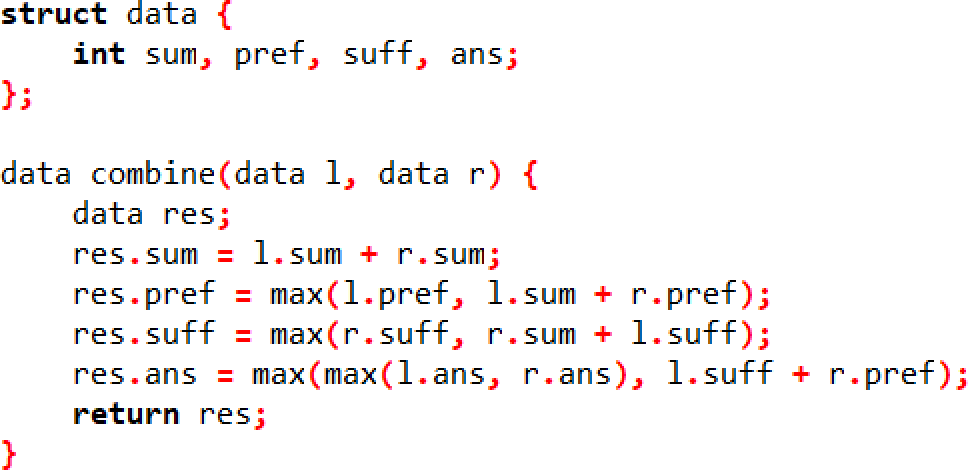
### Tìm các phân đoạn có tổng lớn nhất

Mỗi truy vấn, chúng ta có một mảng a[l … r]; nhiệm vụ phải tìm một phân đoạn a[l’ … r’] điều kiện l <= l’ và r’ <= r và tổng các phần tử của đoạn này là lớn nhất. Như trước đây, chúng ta cũng muốn phải sửa đổi các phần tử riêng lẻ của mảng. Các phần tử của mảng có thể âm và phân đoạn tối ưu có thể trống (ví dụ: nếu tất cả các phần tử đều âm). Lần này chúng ta sẽ lưu trữ bốn giá trị cho mỗi đỉnh: tổng của phân đoạn, tổng tiền tố lớn nhất, tổng hậu tố lớn nhất và tổng của phân đoạn con lớn nhất trong đó. Nói cách khác, đối với mỗi phân đoạn của cây phân đoạn, câu trả lời đã được tính toán trước.

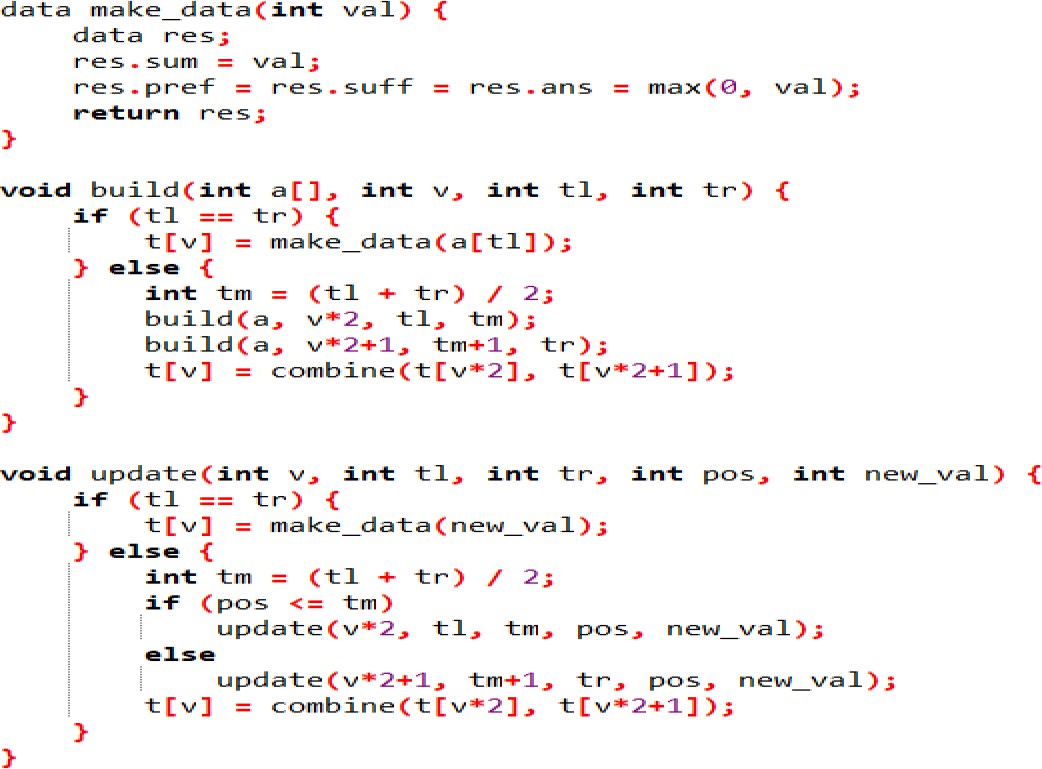
Làm thế nào để xây dựng một cây với dữ liệu như vậy? Một lần nữa, chúng ta tính toán theo kiểu đệ quy: trước tiên chúng ta tính toán tất cả bốn giá trị cho con trái và con phải, sau đó kết hợp các giá trị đó để lưu trữ bốn giá trị cho đỉnh hiện tại. Lưu ý câu đáp án cho đỉnh hiện tại là:

* + - Đáp án con trái, nghĩa là phân đoạn tối ưu được đặt hoàn toàn vào phân đoạn của con trái
    - Đáp án của con phải, có nghĩa là phân khúc tối ưu hoàn toàn được đặt trong phân khúc của con bên phải
    - Tổng của tổng hậu tố tối đa của con trái và tổng tiền tố tối đa của con bên phải, có nghĩa là phân đoạn tối ưu giao với cả hai con.

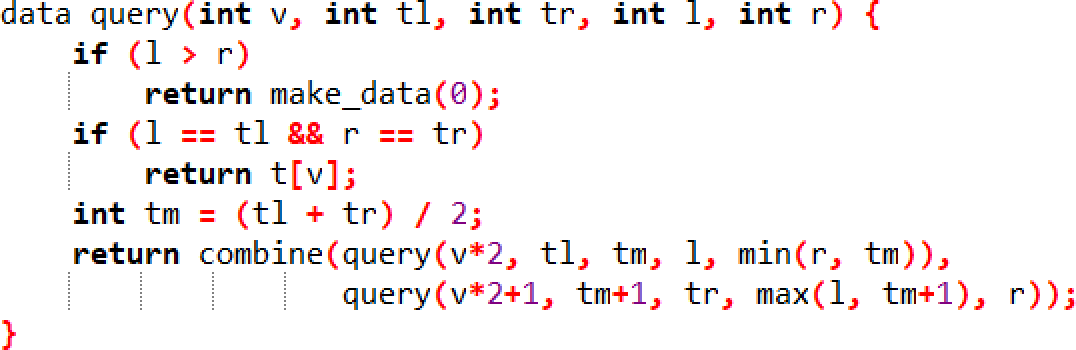
Do đó, đáp án cho đỉnh hiện tại là giá trị lớn nhất trong ba giá trị này. Việc tính toán tổng tiền tố/hậu tố lớn nhất còn dễ dàng hơn. Đây là việc thực hiện các **combine** hàm chỉ nhận dữ liệu từ con trái và con phải, trả về dữ liệu của đỉnh hiện tại.



Sử dụng hàm **combine** dễ dàng để xây dựng cây phân đoạn. Chúng ta có thể triển khai nó theo cách giống như những lần triển khai trước đó. Để khởi tạo các đỉnh lá ta tạo thêm hàm phụ **make\_data** sẽ trả về một **data** đối tượng chứa thông tin của một giá trị duy nhất.



Giờ chỉ còn làm thế nào để tính toán câu trả lời cho một truy vấn. Để trả lời, chúng ta duyệt cây như trước, chia truy vấn thành nhiều phân đoạn trùng với các phân đoạn của cây phân đoạn và kết hợp các câu trả lời trong đó thành một câu trả lời duy nhất cho truy vấn. Công việc hoàn toàn giống như trong cây phân đoạn đơn giản, nhưng thay vì tính tổng/nhỏ nhất/lớn nhất các giá trị, chúng ta sử dụng hàm **combine**.



### Lưu toàn bộ mảng con ở một đỉnh

Đây là một bài toán khác, vì tại mỗi đỉnh của cây phân đoạn chúng ta không lưu trữ thông tin về phân đoạn tương ứng ở dạng nén (tổng, nhỏ nhất, lớn nhất, ...), mà lưu trữ tất cả các phần tử của phân đoạn. Do đó, gốc của cây phân đoạn sẽ lưu trữ tất cả các phần tử của mảng, đỉnh con bên trái sẽ lưu trữ nửa đầu của mảng, đỉnh bên phải lưu trữ nửa sau, v.v.

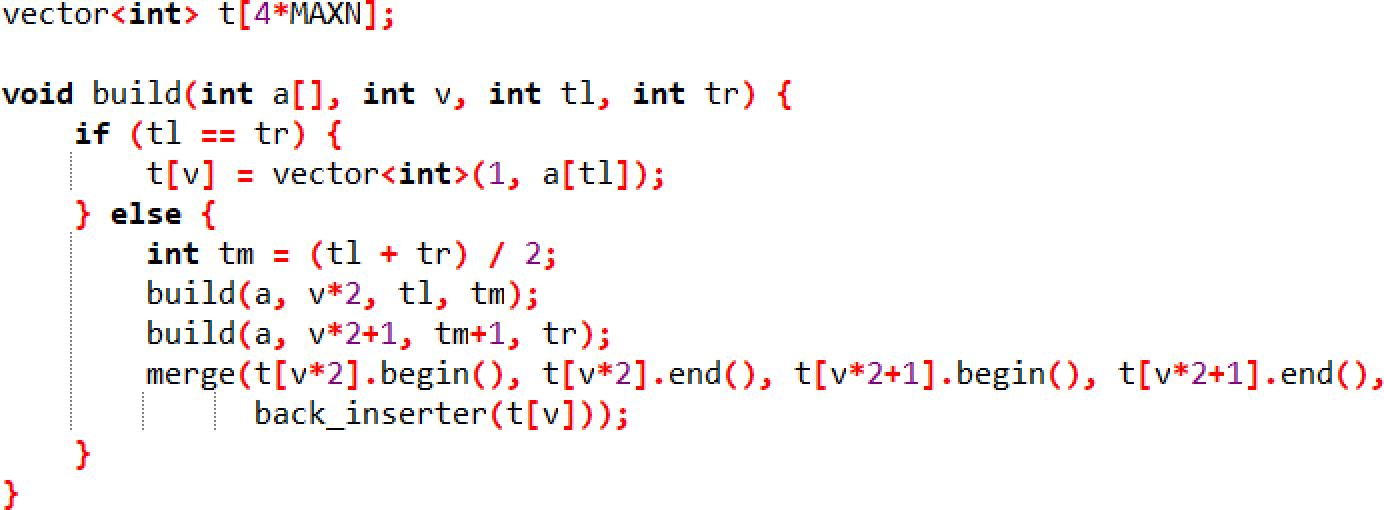
Trong kỹ thuật này, chúng ta lưu trữ các phần tử theo thứ tự được sắp xếp. Trong các phiên bản phức tạp hơn, các phần tử không được lưu trữ trong danh sách mà được lưu trữ trong các cấu trúc dữ liệu nâng cao hơn (set, map, ...). Nhưng tất cả các phương pháp này đều có một điểm chung là mỗi đỉnh đều yêu cầu bộ nhớ tuyến tính (tức là tỷ lệ thuận với độ dài của đoạn tương ứng).

Câu hỏi tự nhiên đầu tiên khi xem xét các cây phân đoạn này là bộ nhớ. Theo trực giác điều này có thể trông giống như O(n2) bộ nhớ, nhưng cây hoàn chỉnh sẽ chỉ cần O(nlogn) bộ nhớ. Bởi vì mỗi phần tử của mảng rơi vào O(log n) các đoạn (chú ý chiều cao của cây là O(logn)).

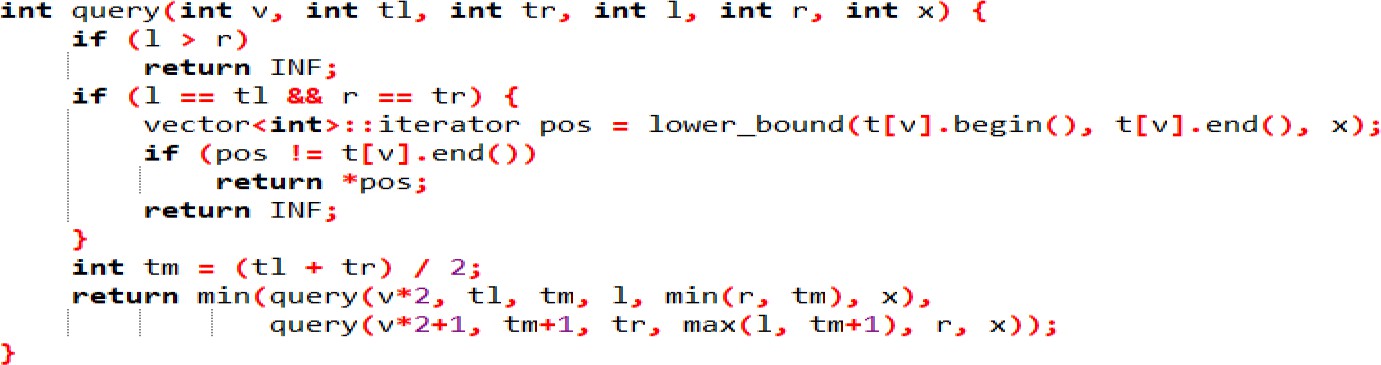
### Tìm số nhỏ nhất lớn hơn hoặc bằng một số được chỉ định, không có truy vấn sửa đổi

Các truy vấn có dạng sau: với ba số đã cho (l, r, x) chúng ta phải tìm số lượng tối thiểu trong phân đoạn a[l … r] lớn hơn hoặc bằng x.

Chúng ta xây dựng một cây phân đoạn. Trong mỗi đỉnh, chúng ta lưu trữ một danh sách được sắp xếp gồm tất cả các số xuất hiện trong đoạn tương ứng như mô tả ở trên. Như thường lệ, chúng ta tiếp cận vấn đề này theo cách đệ quy: giả sử danh sách con trái và con phải đã được xây dựng rồi và chúng ta muốn xây dựng danh sách cho đỉnh hiện tại. Từ ý tưởng này, thao tác giờ đây đơn giản và có thể được thực hiện trong thời gian tuyến tính: Chúng ta chỉ cần kết hợp hai danh sách đã sắp xếp thành một, có thể thực hiện bằng cách lặp qua chúng bằng hai con trỏ. C++ STL đã cài đặt thuật toán

này. Do cấu trúc này của cây phân đoạn và những điểm tương đồng với thuật toán sắp xếp trộn nên cấu trúc dữ liệu này còn thường được gọi là "Cây sắp xếp hợp nhất".

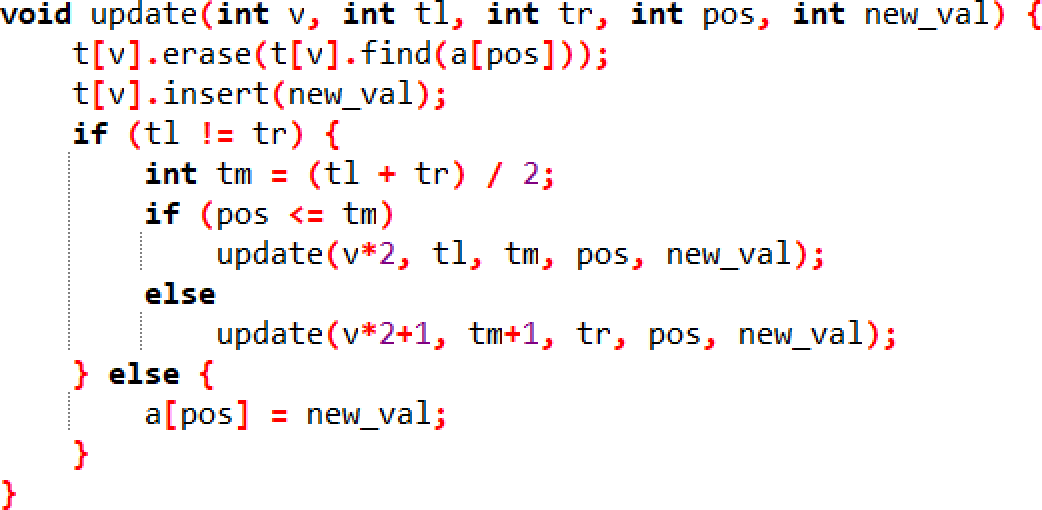
Chúng ta đã biết rằng cây phân đoạn được xây dựng theo cách này sẽ yêu cầu O(nlogn) bộ nhớ. Và nhờ việc triển khai này, việc xây dựng nó cũng mất O(nlogn), mỗi danh sách đều được xây dựng theo thời gian tuyến tính theo kích thước của nó. Bây giờ hãy xem xét câu trả lời cho truy vấn. Chúng ta chia phân đoạn a[l … r] thành nhiều phân đoạn nhỏ (nhiều nhất là O(log n) đoạn). Bây giờ chúng ta chỉ cần hiểu cách trả lời một truy vấn trên một phân đoạn tương ứng với một số đỉnh của cây. Giải sử, chúng ta đang ở một đỉnh nào đó của cây phân đoạn và muốn tính toán câu trả lời cho truy vấn, tức là tìm số nhỏ nhất lớn hơn hoặc bằng một số đã cho x. Vì đỉnh chứa danh sách các phần tử theo thứ tự được sắp xếp nên chúng ta chỉ cần thực hiện tìm kiếm nhị phân trên danh sách này và trả về số đầu tiên, lớn hơn hoặc bằng x. Do đó, câu trả lời cho truy vấn trong một đoạn của cây sẽ là O(log n) và toàn bộ truy vấn được xử lý trong O(log2n).



### Tìm số nhỏ nhất lớn hơn hoặc bằng một số được chỉ định, với các truy vấn sửa đổi

Bài toán này chúng ta muốn thực hiện chính xác điều này: một truy vấn sửa đổi sẽ thực hiện nhiệm vụ a[i] = y. Cách giải cũng tương tự như lời giải của bài toán trên nhưng thay vì liệt kê ở mỗi đỉnh của cây phân đoạn, chúng ta sẽ lưu trữ một danh sách

cân bằng cho phép tìm kiếm nhanh số, xóa số và chèn số mới. Vì mảng có thể chứa một số lặp lại nên lựa chọn tối ưu là cấu trúc dữ liệu multiset.

Việc xây dựng cây phân đoạn như vậy được thực hiện gần giống như trong bài toán trước, chỉ khác là bây giờ chúng ta cần kết hợp multiset và danh sách không được sắp xếp. Điều này dẫn đến thời gian thực hiện O(nlog2n) (việc hợp nhất hai cây đỏ đen có thể được thực hiện theo thời gian tuyến tính, nhưng C++ STL không đảm bảo độ phức tạp của thời gian này). Hàm lower\_bound của multiset sẽ được gọi (std::lower\_bound chỉ hoạt động O(log n) nếu được sử dụng với các vòng lặp ngẫu nhiên). Cuối cùng là yêu cầu sửa đổi. Để xử lý, chúng ta phải duyệt cây và sửa đổi tất cả multiset từ các phân đoạn tương ứng có chứa phần tử bị ảnh hưởng. Chúng ta chỉ cần xóa giá trị cũ của phần tử này (nhưng chỉ một lần xuất hiện) và chèn giá trị mới.

Việc xử lý truy vấn sửa đổi này cũng mất O(log2n) thời gian.

## Kỹ thuật lan truyền lười (Lazy Propagation)

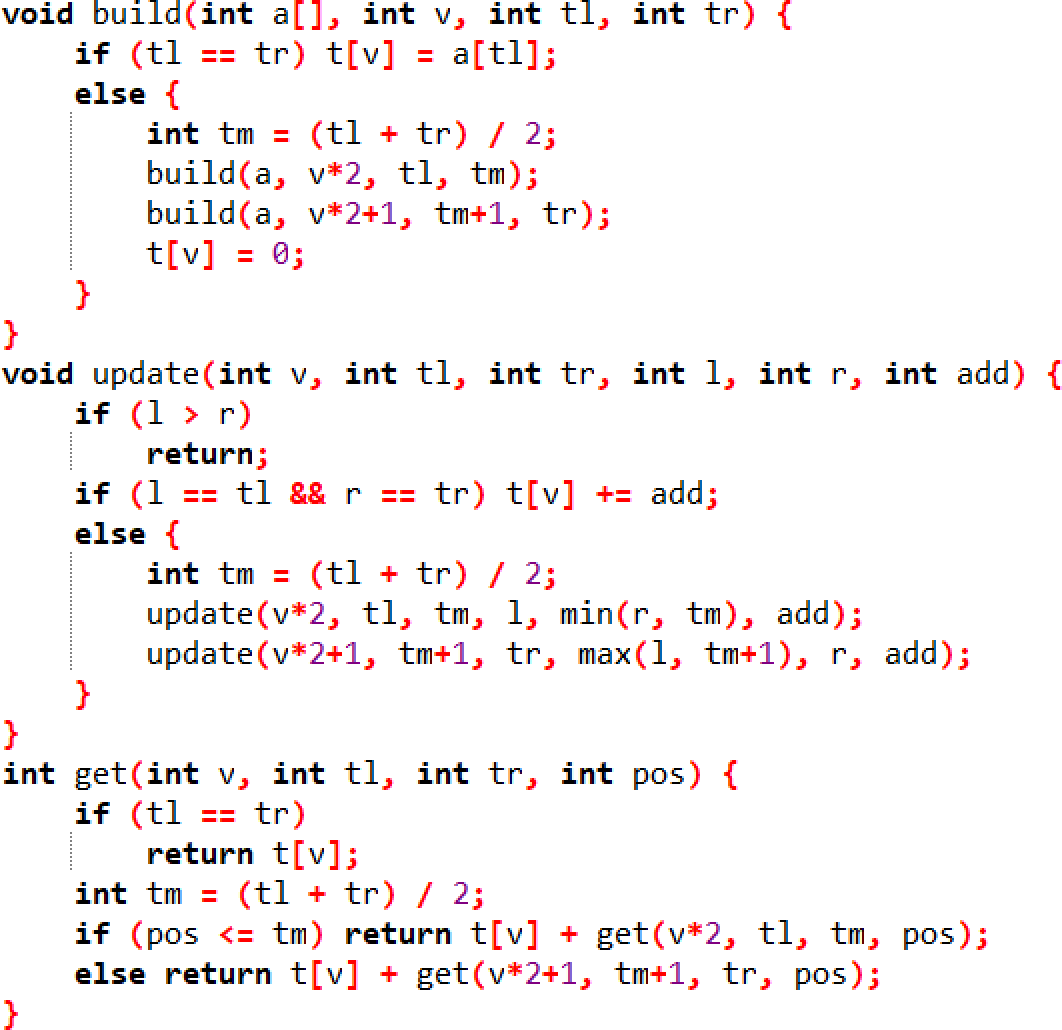
Tất cả các bài toán trong các phần trên đều đề cập về các truy vấn sửa đổi chỉ ảnh hưởng đến một phần tử duy nhất của mảng. Tuy nhiên, cây phân đoạn cho phép áp dụng các truy vấn sửa đổi cho toàn bộ phân đoạn gồm các phần tử liền kề và thực hiện truy vấn trong cùng một thời điểm O(logn).

### Cộng trên các phân đoạn

Chúng ta tiếp cận các vấn đề ở dạng đơn giản nhất: truy vấn sửa đổi sẽ thêm một số x tới tất cả các số trong phân đoạn a[l … r]. Truy vấn thứ hai, mục đích chúng ta trả lời và hỏi về giá trị của a[i].

Để truy vấn cộng hiệu quả, chúng ta lưu trữ tại mỗi đỉnh trong cây phân đoạn số lượng nên thêm vào tất cả các số trong phân đoạn tương ứng. Ví dụ nếu truy vấn "thêm 3 vào toàn bộ mảng a[0 … n-1]”, chúng ta đặt số 3 vào gốc cây. Nói chung ta phải đặt số này thành nhiều phân đoạn, tạo thành một phân vùng của phân đoạn truy vấn. Như vậy chúng ta không phải thay đổi tất cả O(n) giá trị mà chỉ O(logn).

Nếu có một truy vấn hỏi giá trị hiện tại của một mảng cụ thể, thì chỉ việc duyệt cây và cộng tất cả các giá trị được tìm thấy trong quá trình thực hiện là đủ.



### Gán trên các phân đoạn

Giả sử bây giờ truy vấn sửa đổi yêu cầu gán từng phần tử của một phân đoạn nhất định a[l … r] đến một giá trị nào đó p. Như truy vấn thứ hai, chúng ta sẽ xem xét lại việc đọc giá trị của mảng a[i]. Để thực hiện truy vấn sửa đổi này trên toàn bộ phân đoạn, ta phải lưu trữ tại mỗi đỉnh của cây phân đoạn xem phân đoạn tương ứng có được bao phủ hoàn toàn bằng cùng một giá trị hay không. Điều này cho phép chúng ta thực hiện cập nhật "lười ": thay vì thay đổi tất cả các phân đoạn trong cây bao gồm

phân đoạn truy vấn, chúng ta chỉ thay đổi một số và giữ nguyên những phân đoạn khác. Một đỉnh được đánh dấu sẽ có nghĩa là mọi phần tử của đoạn tương ứng được gán cho giá trị đó và trên thực tế, cây con hoàn chỉnh cũng chỉ nên chứa giá trị này. Theo một nghĩa nào đó, chúng ta lười biếng và trì hoãn việc ghi giá trị mới vào tất cả các đỉnh đó. Chúng ta có thể thực hiện công việc tẻ nhạt này sau nếu cần thiết. Vì vậy, sau khi truy vấn sửa đổi được thực thi, một số phần của cây trở nên không liên quan - một số sửa đổi vẫn chưa được thực hiện trong đó.

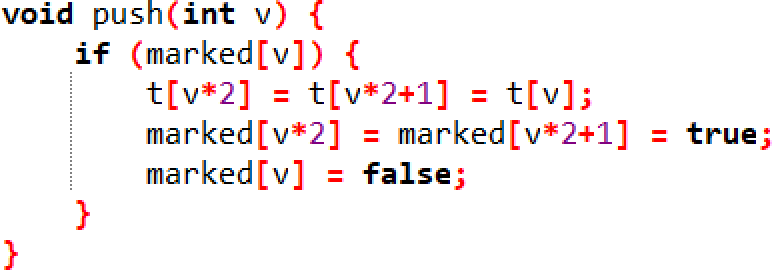
Ví dụ, nếu truy vấn sửa đổi "*gán một số cho toàn bộ mảng a[0 … n-1]*” được thực thi, trong cây phân đoạn chỉ có một thay đổi duy nhất được thực hiện - số được đặt ở gốc của cây và đỉnh này được đánh dấu. Các phân đoạn còn lại không thay đổi, mặc dù trên thực tế, số đó phải được đặt trong toàn bộ cây.

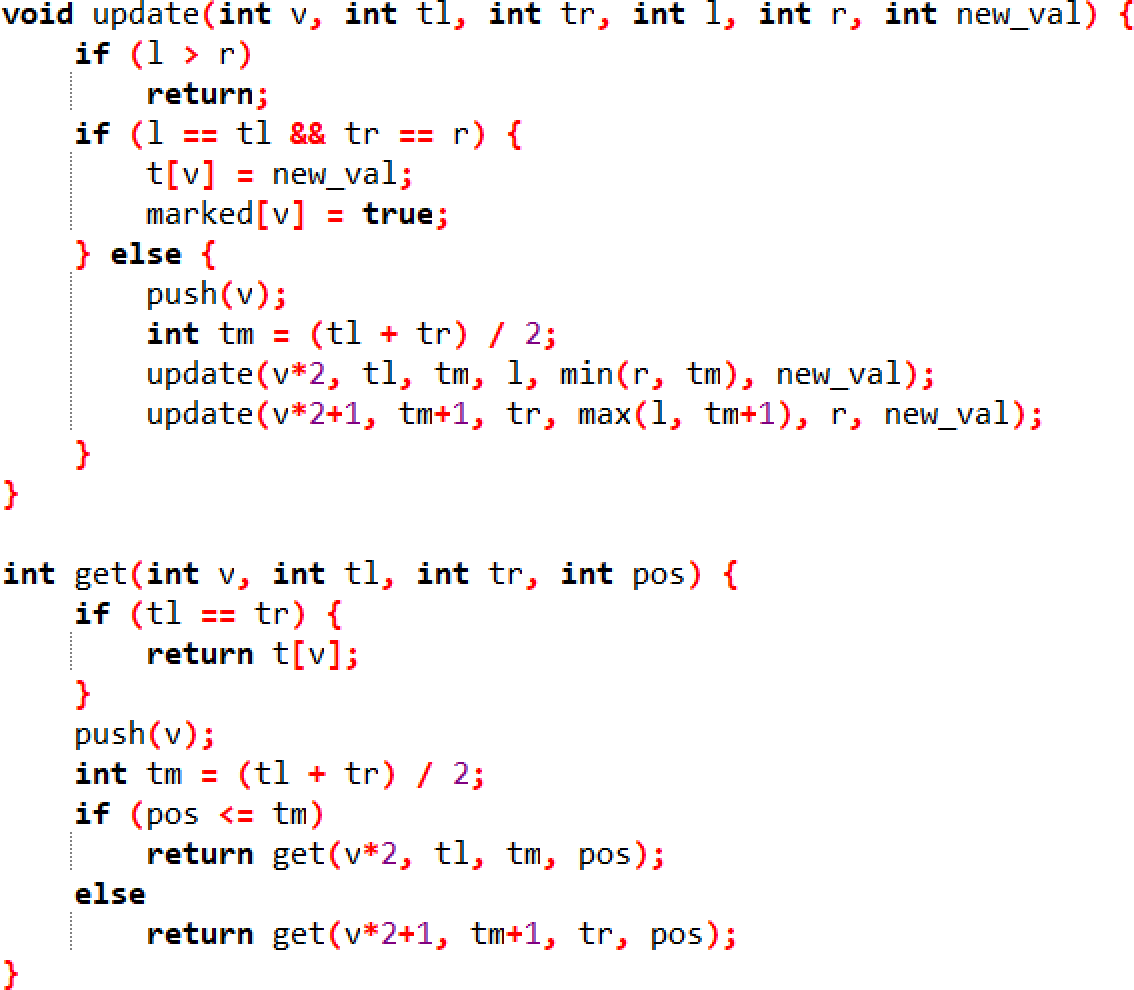
Giả sử bây giờ truy vấn sửa đổi thứ hai yêu cầu một nửa mảng a[0 … n/2] được gán với một số số khác. Để xử lý truy vấn này, chúng ta phải gán từng phần tử trong toàn bộ phần tử con bên trái của đỉnh gốc với số đó. Nhưng trước khi làm điều này, chúng ta phải sắp xếp đỉnh gốc. Ở đây là nửa bên phải của mảng vẫn phải được gán cho giá trị của truy vấn đầu tiên và hiện tại không có thông tin nào cho nửa bên phải được lưu trữ.

Cách giải quyết là đẩy thông tin của gốc tới các con của nó, tức là nếu gốc của cây được gán một số nào đó thì ta gán số đó cho các đỉnh con trái và phải của cây và bỏ đánh dấu của gốc đó. Sau đó, chúng ta có thể gán giá trị mới cho con bên trái mà không làm mất bất kỳ thông tin cần thiết nào.

Tóm lại, đối với bất kỳ truy vấn nào (truy vấn sửa đổi hoặc đọc) trong quá trình di chuyển dọc theo cây, chúng tôi phải luôn đẩy thông tin từ đỉnh hiện tại vào cả hai đỉnh con của nó. Chúng ta có thể hiểu điều này theo cách mà khi đi xuống cây, chúng ta áp dụng các sửa đổi bị trì hoãn, nhưng chính xác ở mức cần thiết (để không làm giảm độ phức tạp của O(log n)).

Để thực hiện chúng ta cần thực hiện một hàm **push** , hàm này sẽ nhận đỉnh hiện tại và nó sẽ đẩy thông tin về đỉnh tới cả hai đỉnh con của nó. Chúng ta sẽ gọi hàm này ở đầu các hàm truy vấn.





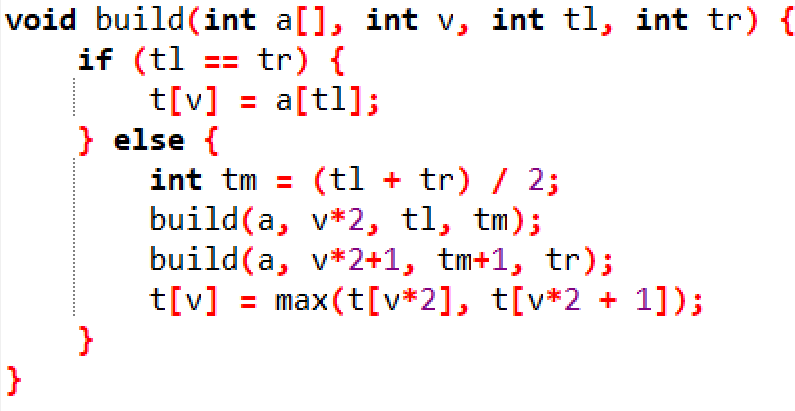
Lưu ý hàm **get** cũng có thể được thực hiện theo một cách khác: không thực hiện cập nhật chậm trễ mà trả về ngay giá trị t[v] nếu như marked[v] là đúng.

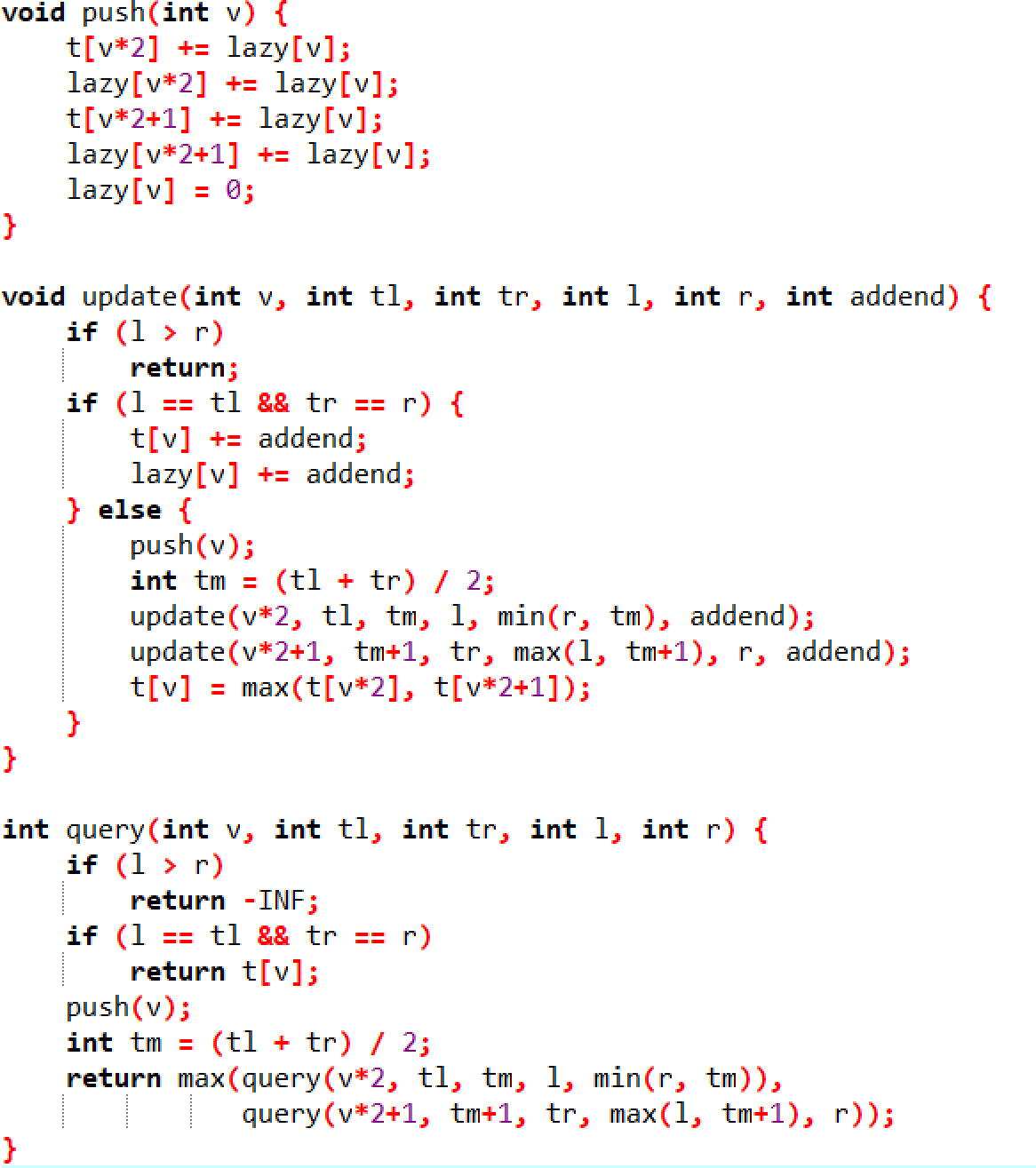
### Cộng trên phân đoạn, truy vấn tìm giá trị lớn nhất

Bây giờ truy vấn sửa đổi là thêm một số vào tất cả các phần tử trong một phạm vi và truy vấn đọc là tìm giá trị lớn nhất trong một phạm vi.

Vì vậy, đối với mỗi đỉnh của cây phân đoạn, chúng ta phải lưu trữ tối đa phân đoạn tương ứng. Với mục đích này, chúng ta tiếp tục lưu trữ một giá trị bổ sung cho mỗi đỉnh. Trong giá trị này, chúng ta lưu trữ các phần bổ sung mà chúng ta chưa

truyền tới các đỉnh con. Trước khi đi qua một đỉnh con, chúng ta gọi **push** và truyền giá trị cho cả hai con. Chúng ta phải làm điều này trong cả hai hàm **update** và **query.**



****

## Khái quát hóa đến các chiều cao hơn

Cây phân đoạn có thể được khái quát khá tự nhiên đến các kích thước cao hơn. Nếu trong trường hợp một chiều, chúng ta chia các chỉ mục của mảng thành các phân đoạn thì trong trường hợp hai chiều, chúng ta tạo một cây phân đoạn thông thường đối

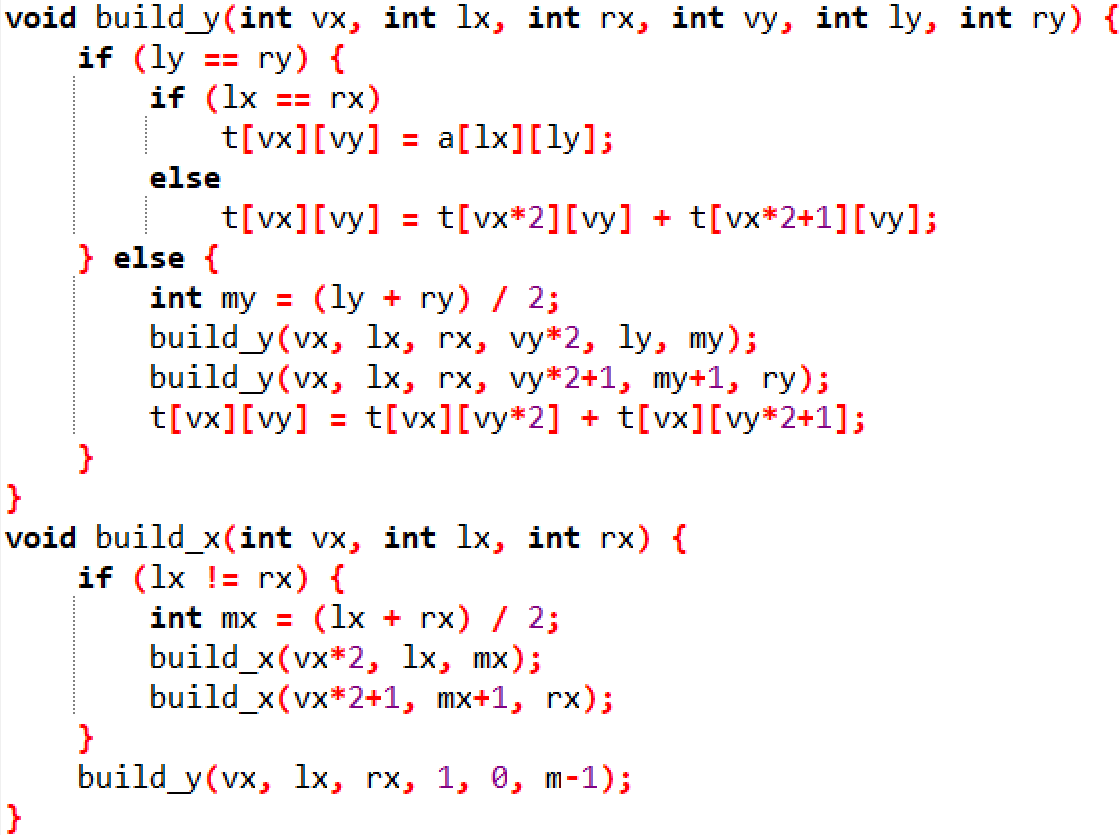
với các chỉ mục đầu tiên và đối với mỗi phân đoạn, chúng ta xây dựng một Cây phân đoạn thông thường đối với các chỉ số đầu tiên. chỉ số thứ hai.

### Cây phân đoạn 2D đơn giản

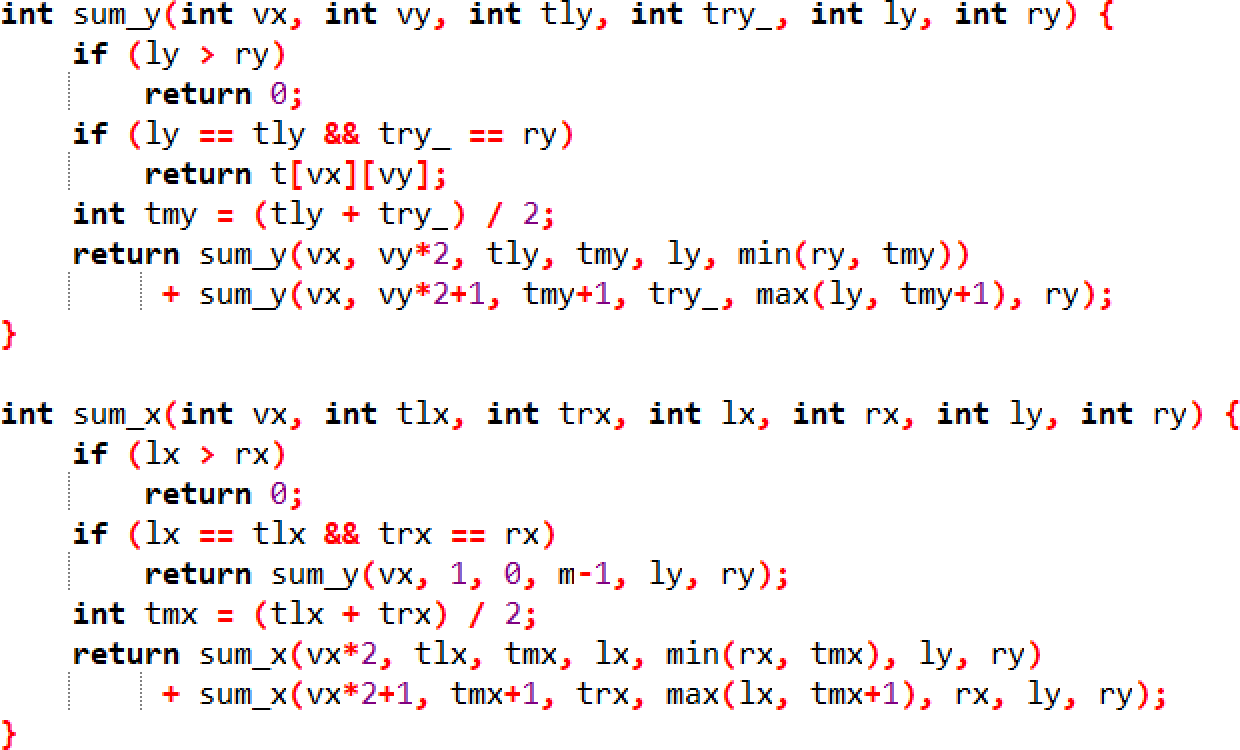
Cho một ma trận a[0 … n-1, 0 … m-1] và yêu cầu phải tìm tổng (hoặc nhỏ nhất/lớn nhất) trên một số ma trận con a[x1 … x2, y1 … y2] cũng như thực hiện sửa đổi các phần tử ma trận riêng lẻ (tức là các truy vấn có dạng a[x][y] = p). Vì vậy, chúng ta xây dựng cây phân đoạn 2D: đầu tiên là cây phân đoạn sử dụng tọa độ (x) và (y).

Để quá trình xây dựng trở nên dễ hiểu hơn, có thể tạm quên rằng ma trận là hai chiều và chỉ để lại tọa độ đầu tiên. Chúng ta sẽ xây dựng cây phân đoạn một chiều thông thường chỉ bằng tọa độ đầu tiên. Nhưng thay vì lưu trữ một số trong một phân đoạn, ta lưu trữ toàn bộ cây phân đoạn: tức là tại thời điểm này, nhớ rằng chúng ta cũng có tọa độ thứ hai; nhưng bởi vì tại thời điểm này tọa độ đầu tiên đã được cố định ở một khoảng nào đó [l … r] , nên ta thực sự làm việc với một dải a[l … r, 0 … m-1] và để làm được điều đó, chúng ta xây dựng cây phân đoạn.

Đây là cách thực hiện việc xây dựng cây phân đoạn 2D. Nó đại diện cho hai khối riêng biệt: việc xây dựng cây phân đoạn dọc theo x điều phối **built\_x** và y điều phối **built\_y.** Đối với các nút lá trong **built\_y** chúng ta phải tách hai trường hợp: khi đoạn hiện tại của tọa độ đầu tiên [tlx … trx] có độ dài bằng 1 và khi có độ dài lớn hơn một. Trong trường hợp đầu tiên, chúng ta chỉ lấy giá trị tương ứng từ ma trận và trong trường hợp thứ hai, chúng ta có thể kết hợp các giá trị của hai cây phân đoạn từ bên trái và bên phải trong tọa độ x.



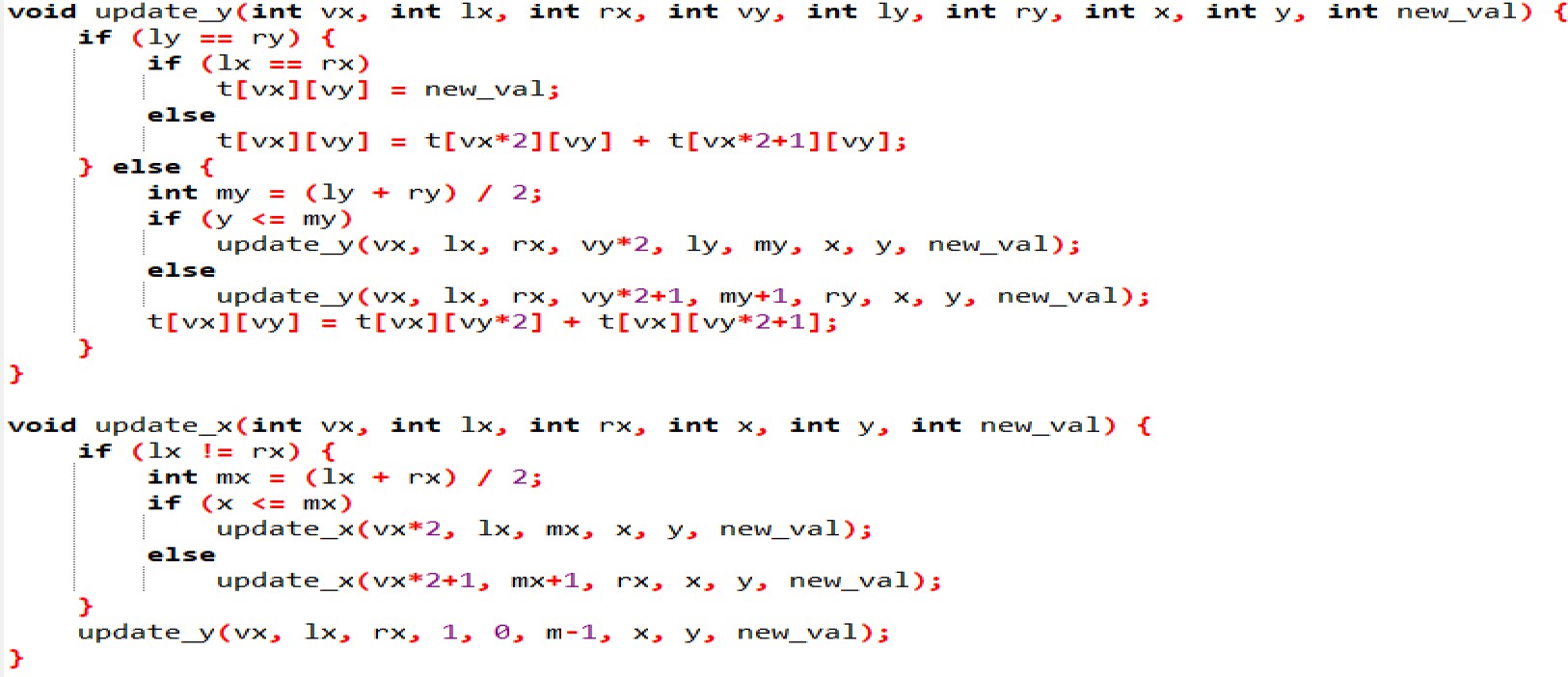
Cây phân đoạn như vậy vẫn sử dụng lượng bộ nhớ tuyến tính, nhưng với hằng số lớn hơn 16\*n\*m. Rõ ràng là quy trình được mô tả built\_x cũng hoạt động trong thời gian tuyến tính.

Bây giờ chúng ta chuyển sang xử lý các truy vấn. Chúng ta sẽ trả lời truy vấn hai chiều bằng cách sử dụng cùng một nguyên tắc: đầu tiên ngắt truy vấn trên tọa độ đầu tiên và sau đó với mỗi đỉnh đạt được, chúng ta gọi cây phân đoạn tương ứng của tọa độ thứ hai.

Hàm này hoạt động trong O(log(n) \* log(m)), vì lần đầu tiên nó đi xuống cây ở tọa độ đầu tiên và với mỗi đỉnh đi qua trong cây, nó thực hiện một truy vấn trong cây phân đoạn tương ứng dọc theo tọa độ thứ hai.

Cuối cùng xem xét truy vấn sửa đổi. Chúng ta muốn tìm hiểu cách sửa đổi cây

phân đoạn theo sự thay đổi giá trị của một số phần tử a[x][y] = p. Rõ ràng là những thay đổi sẽ chỉ xảy ra ở các đỉnh của cây phân đoạn đầu tiên bao phủ tọa độ x (và đó sẽ là O(log n)) và đối với cây phân đoạn tương ứng với chúng, các thay đổi sẽ chỉ xảy ra ở các đỉnh bao phủ tọa độ y (và đó sẽ là O(log m)). Do đó, việc triển khai sẽ không khác lắm so với trường hợp một chiều, chỉ khác là bây giờ chúng ta lần đầu tiên đi xuống tọa độ đầu tiên và sau đó là tọa độ thứ hai.



### Nén cây phân đoạn 2D

Giả sử bài toán như sau: Có n các điểm trên mặt phẳng được cho bởi tọa độ của chúng (xi, yi) và truy vấn dạng “*đếm số điểm nằm trong hình chữ nhật ((x1, y1`)), (x2, y2)*”. Rõ ràng là trong trường hợp này, việc xây dựng cây phân đoạn hai chiều với O(n2) phần tử. Phần lớn bộ nhớ này sẽ bị lãng phí vì mỗi điểm chỉ có thể được đưa vào O(logn) các phân đoạn của cây dọc theo tọa độ đầu tiên và do đó tổng kích thước "hữu dụng" của tất cả các phân đoạn cây trên tọa độ thứ hai là O(nlogn).

Vì vậy, chúng ta tiến hành như sau: tại mỗi đỉnh của cây phân đoạn đối với tọa độ đầu tiên, chúng ta lưu trữ cây phân đoạn chỉ được tạo bởi các tọa độ thứ hai xuất hiện trong phân đoạn hiện tại của tọa độ đầu tiên. Nói cách khác, khi xây dựng cây phân đoạn bên trong một số đỉnh có chỉ số vx và ranh giới tlx và trx, ta chỉ xét những điểm rơi vào khoảng x thuộc [tlx, trx] và xây dựng cây phân đoạn chỉ bằng cách sử dụng chúng.

Do đó, chúng ta sẽ biết được rằng mỗi cây phân đoạn trên tọa độ thứ hai sẽ

chiếm chính xác nhiều bộ nhớ nhất có thể. Kết quả là tổng dung lượng bộ nhớ sẽ giảm xuống O(nlogn). Ta vẫn có thể trả lời các câu hỏi trong O(log2n) và chỉ cần thực hiện tìm kiếm nhị phân trên tọa độ thứ hai, nhưng điều này sẽ không làm giảm độ phức tạp.

Nhưng các truy vấn sửa đổi sẽ không thể thực hiện được với cấu trúc này: trên thực tế, nếu một điểm mới xuất hiện, chúng ta phải thêm một phần tử mới vào giữa một số cây phân đoạn dọc theo tọa độ thứ hai, điều này không thể thực hiện được một cách hiệu quả.

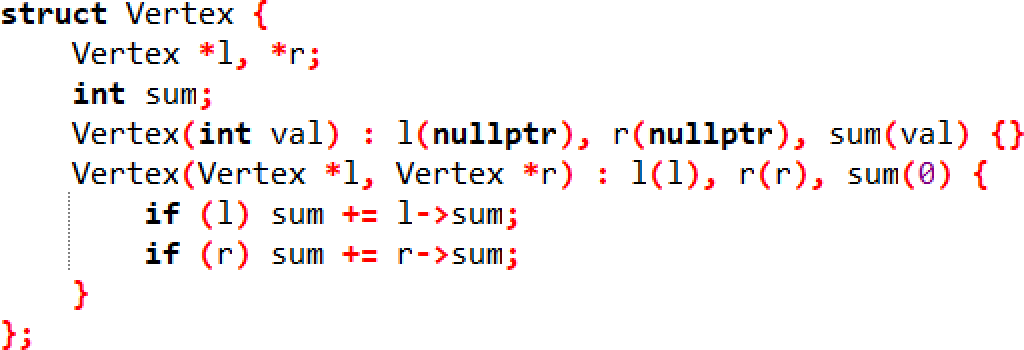
Tóm lại, chúng ta lưu ý rằng cây phân đoạn hai chiều được thu gọn theo cách được mô tả trên thực tế trở nên tương đương với việc sửa đổi cây phân đoạn một chiều (xem [Lưu toàn bộ mảng con trong mỗi đỉnh](https://cp-algorithms.com/data_structures/segment_tree.html#saving-the-entire-subarrays-in-each-vertex)). Cụ thể, cây phân đoạn hai chiều chỉ là trường hợp đặc biệt lưu trữ một mảng con ở mỗi đỉnh của cây. Theo đó, nếu bạn quyết định từ bỏ cây phân đoạn hai chiều do không thể thực hiện truy vấn, bạn nên thử thay thế Cây phân đoạn lồng nhau bằng một số cấu trúc dữ liệu mạnh hơn, ví dụ như cây Descartes.

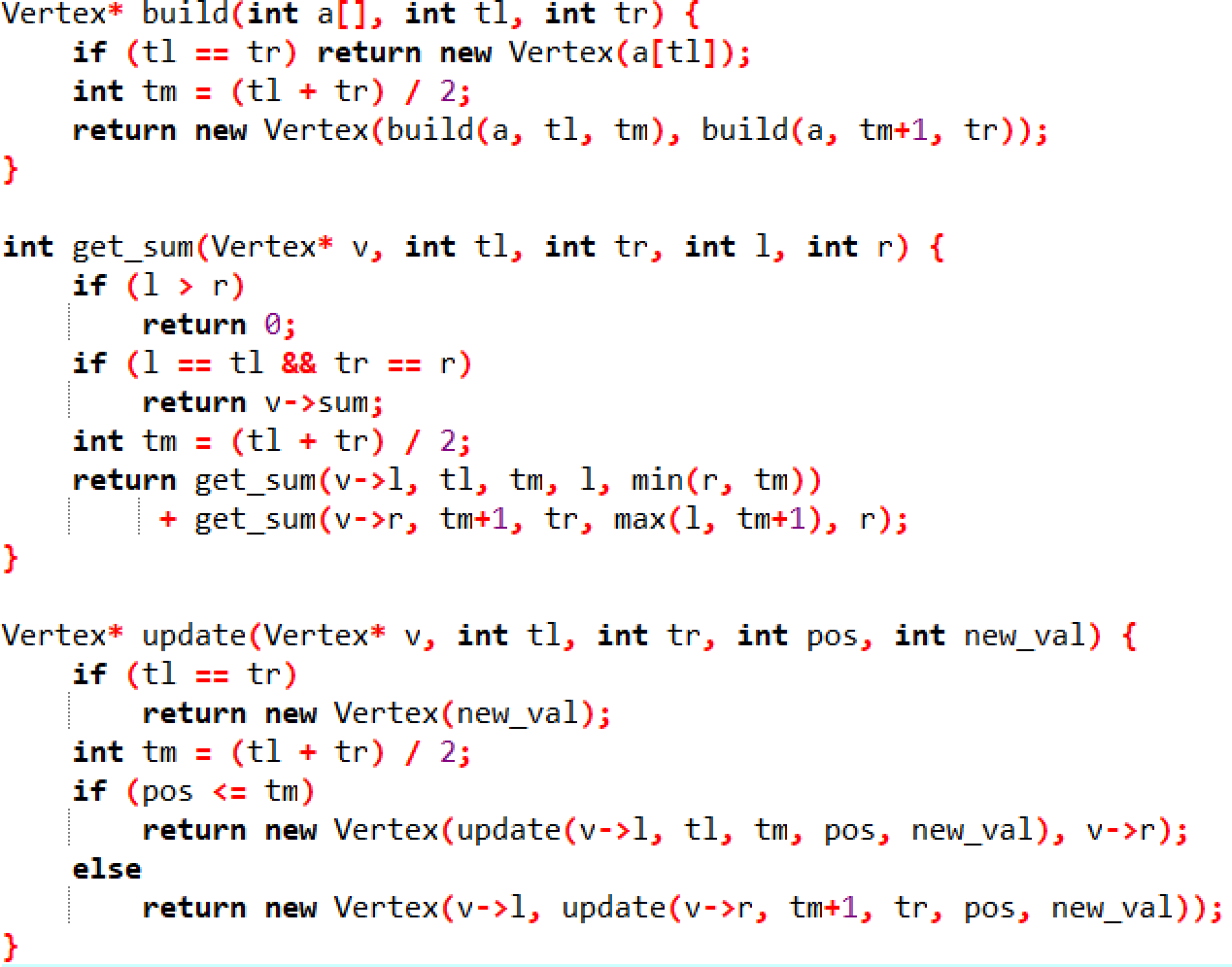
### Cây phân đoạn liên tục

Cấu trúc dữ liệu liên tục là cấu trúc dữ liệu ghi nhớ trạng thái trước đó cho mỗi lần sửa đổi. Điều này cho phép truy cập bất kỳ phiên bản nào của cấu trúc dữ liệu này mà chúng ta quan tâm và thực hiện truy vấn trên đó. Cây phân đoạn là cấu trúc dữ liệu có thể được chuyển thành cấu trúc dữ liệu liên tục một cách hiệu quả (cả về thời gian và mức tiêu thụ bộ nhớ). Chúng ta muốn tránh sao chép toàn bộ cây trước mỗi lần sửa đổi và chúng ta không muốn mất đi O(log n) thời gian để trả lời các truy vấn đoạn.

Trên thực tế, bất kỳ yêu cầu thay đổi nào trong cây phân đoạn đều dẫn đến thay đổi dữ liệu của O(log n) các đỉnh dọc theo đường đi bắt đầu từ gốc. Vì vậy, nếu chúng ta lưu trữ Segment Tree bằng cách sử dụng các con trỏ (tức là một đỉnh lưu trữ các con trỏ ở các đỉnh con bên trái và bên phải), thì khi thực hiện truy vấn sửa đổi, chúng ta chỉ cần tạo các đỉnh mới thay vì thay đổi các đỉnh có sẵn. Các đỉnh không bị ảnh hưởng bởi truy vấn sửa đổi vẫn có thể được sử dụng bằng cách trỏ con trỏ tới các đỉnh cũ. Vì vậy, đối với một truy vấn sửa đổi O(log n) các đỉnh mới sẽ được tạo, bao gồm một đỉnh gốc mới của Cây phân đoạn và toàn bộ phiên bản trước đó của cây có gốc ở đỉnh gốc cũ sẽ không thay đổi.

Ví dụ triển khai cho cây phân đoạn đơn giản nhất: khi chỉ có một truy vấn yêu cầu tính tổng và truy vấn sửa đổi các phần tử đơn lẻ.





Đối với mỗi sửa đổi của cây phân đoạn, chúng ta sẽ nhận được một đỉnh gốc mới. Để nhanh chóng chuyển đổi giữa hai phiên bản khác nhau của cây phân đoạn, chúng ta cần lưu trữ gốc này trong một mảng. Để sử dụng một phiên bản cụ thể của cây phân đoạn, chúng ta chỉ cần gọi truy vấn bằng cách sử dụng đỉnh gốc thích hợp. Với cách tiếp cận được mô tả ở trên, hầu hết mọi cây phân đoạn đều có thể được chuyển thành cấu trúc dữ liệu ổn định.

### Tìm kiếm số nhỏ nhất thứ k trong phạm vi

Lần này chúng ta phải trả lời các truy vấn có dạng "*Tìm phần tử nhỏ nhất thứ k trong phạm vi a[l … r]*”. Truy vấn này có thể được trả lời bằng cách sử dụng tìm kiếm nhị phân và cây sắp xếp hợp nhất, nhưng độ phức tạp về thời gian cho một truy vấn sẽ là O(log3n). Chúng ta sẽ tiếp cận giải bằng cách sử dụng cây phân đoạn liên tục trong O(log n).

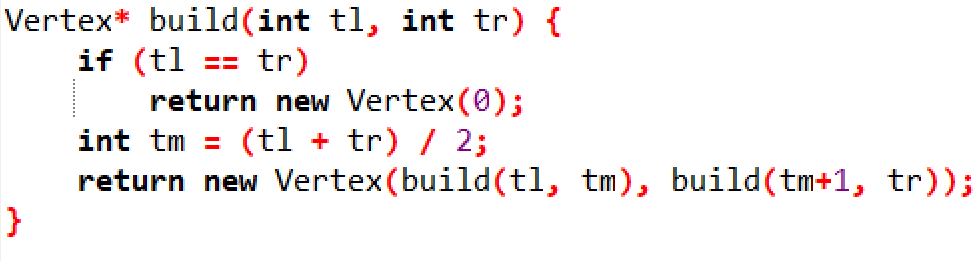
Đầu tiên chúng ta sẽ thảo luận về giải pháp cho một vấn đề đơn giản hơn:

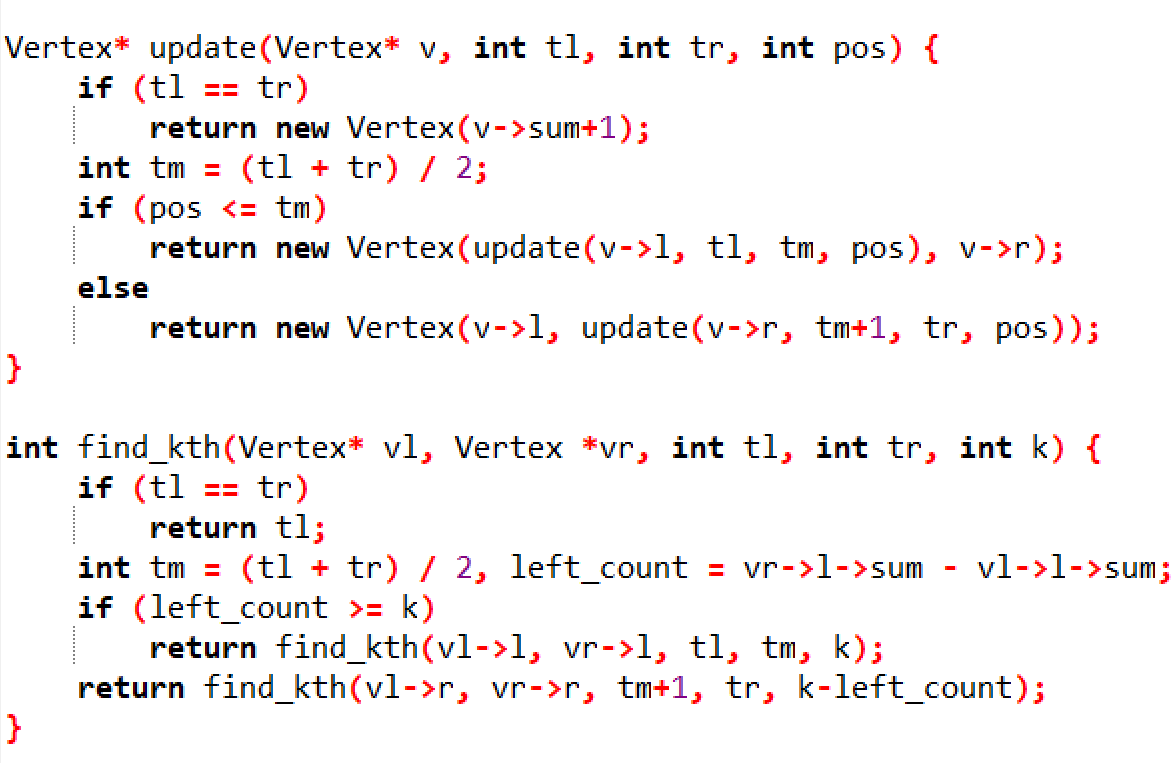
Chúng ta sẽ chỉ xem xét các mảng trong đó các phần tử được giới hạn bởi 0 <= a[i] <

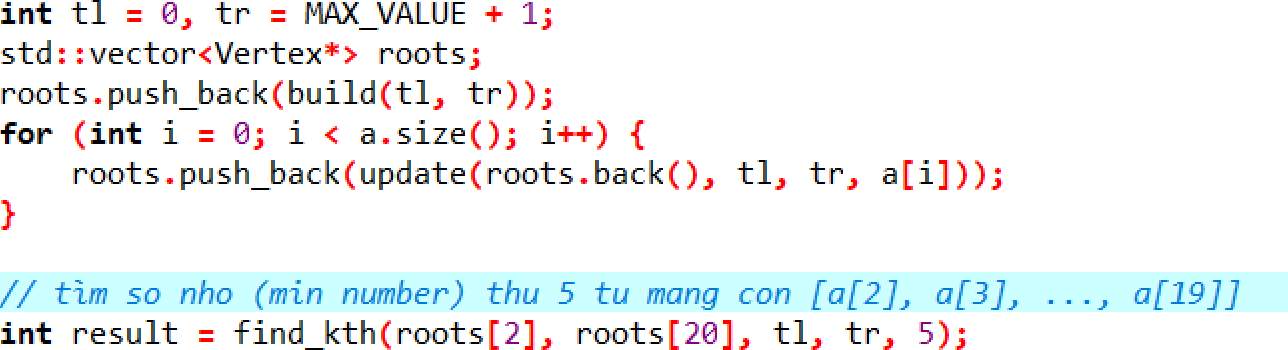
n. Và chúng ta chỉ muốn tìm phần tử thứ k nhỏ nhất trong tiền tố của mảng a[]. Sẽ rất dễ dàng để mở rộng các ý tưởng được phát triển sau này cho các mảng không bị giới hạn và các truy vấn phạm vi không bị giới hạn. Lưu ý rằng chúng ta sẽ sử dụng 1 dựa trên chỉ mục cho a. Chúng ta sẽ sử dụng cây phân đoạn để đếm tất cả các số xuất hiện, tức là trong cây phân đoạn chúng ta sẽ lưu trữ tần số của mảng. Vì vậy các đỉnh lá sẽ lưu trữ tần suất các giá trị 0, 1, … n-1 sẽ xuất hiện trong mảng và các đỉnh khác lưu trữ có bao nhiêu số trong một phạm vi nào đó trong mảng. Nói cách khác, chúng ta tạo cây phân đoạn thông thường với các truy vấn tổng trên tần số của mảng. Nhưng thay vì tạo ra tất cả n cây phân đoạn cho mọi tiền tố có thể, chúng ta sẽ tạo một tiền tố liên tục, sẽ chứa cùng một thông tin. Chúng ta sẽ bắt đầu với cây phân đoạn trống (tất cả số lượng sẽ được tính 0) được chỉ bởi root0 và thêm các phần tử a[1], a[2], …, a[n] lần lượt. Đối với mỗi sửa đổi, chúng ta sẽ nhận được một đỉnh gốc mới, gọi là rooti gốc của cây phân đoạn sau khi chèn đoạn đầu tiên i các phần tử của mảng a. Cây phân đoạn bắt nguồn từ rooti sẽ chứa tần suất của tiền tố a[1 … i]. Sử dụng cây phân đoạn này chúng ta có thể tìm thấy trong O(logn) thời gian để tìm vị trí của phần tử thứ k sử dụng kỹ thuật tương tự được thảo luận trong bài toán “*Đếm số 0, tìm kiếm số 0 thứ k*”.

Bây giờ đến phiên bản động của vấn đề. Đầu tiên về hạn chế đối với các truy vấn: Thay vì chỉ thực hiện các truy vấn này qua tiền tố của a chúng ta muốn sử dụng bất kỳ phân đoạn tùy ý a[l … r] . Ở đây chúng ta cần cây phân đoạn biểu thị cho tần số của các phần tử trong phạm vi a[l … r]. Dễ dàng nhận thấy rằng cây phân đoạn như vậy chỉ là sự khác biệt giữa cây phân đoạn có gốc tại rootr và cây phân đoạn bắt nguồn từ rootl-1, tức là mọi đỉnh trong [l … r]. Cây phân đoạn có thể được tính bằng đỉnh của rootr cây trừ đỉnh của rootl-1 cây. Trong việc thực hiện các hàm find\_kth này có thể được xử lý bằng cách truyền hai con trỏ đỉnh và tính đếm/tổng của đoạn hiện tại dưới dạng chênh lệch của hai đếm/tổng của các đỉnh.

Dưới đây là những sửa đổi hàm **built, update** và **find\_kth**:





Như đã trình bày ở trên, chúng ta cần lưu trữ gốc của cây phân đoạn ban đầu cũng như tất cả các gốc sau mỗi lần cập nhật. Đây là code để xây dựng cây phân đoạn ổn định trên một vector a có các phần tử trong phạm vi [0, MAXVALUE]

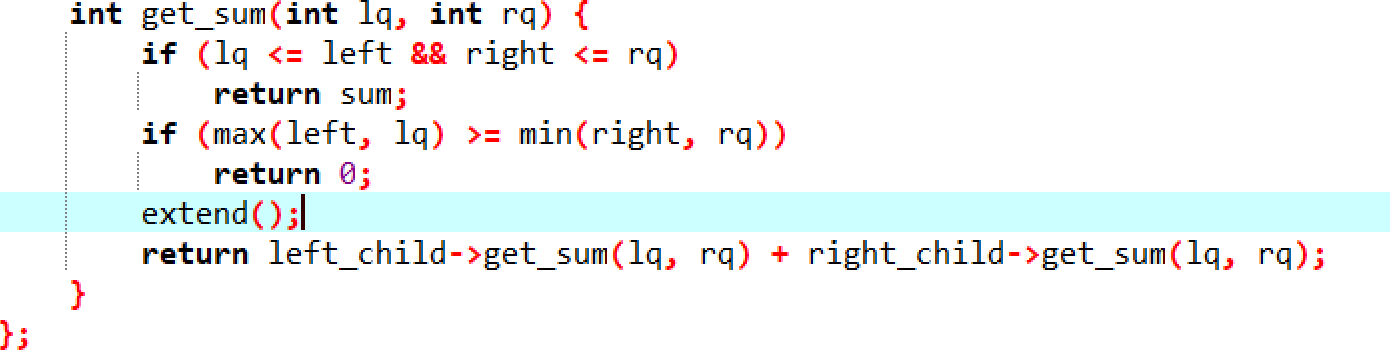
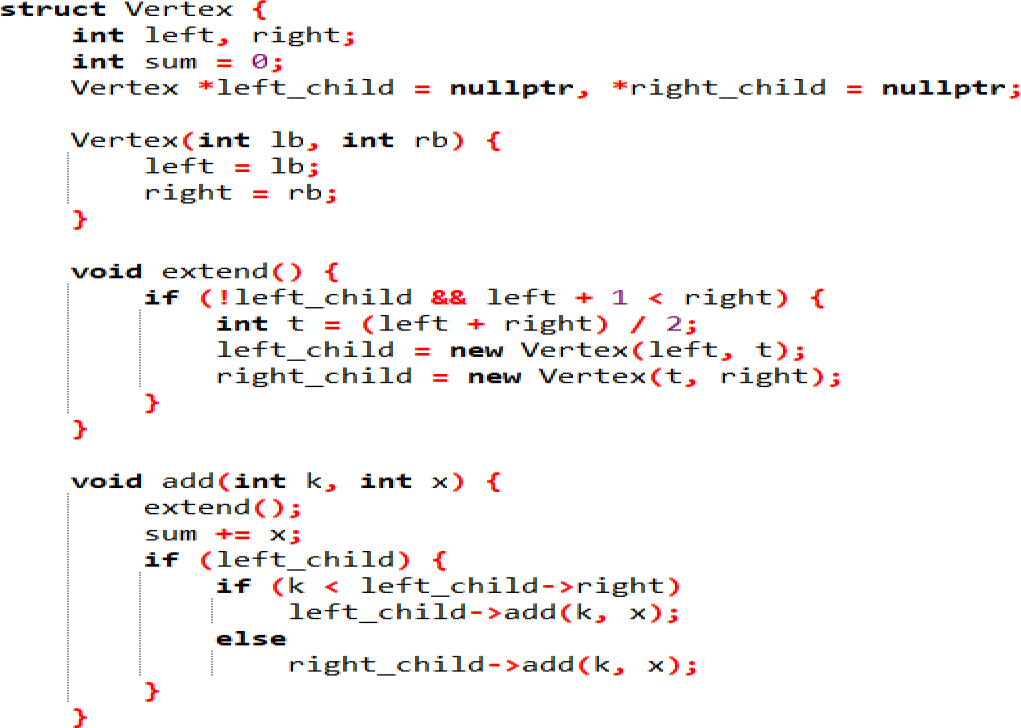
Bây giờ đến các hạn chế đối với các phần tử mảng: Chúng ta thực sự có thể chuyển đổi bất kỳ mảng nào thành mảng như vậy bằng cách nén chỉ mục. Phần tử nhỏ nhất trong mảng sẽ được gán giá trị 0, phần tử nhỏ thứ hai có giá trị 1, v.v. Thật dễ dàng để tạo ra các bảng tra cứu (ví dụ sử dụng map ), chuyển đổi một giá trị thành chỉ mục của nó và ngược lại trong O(log n) thời gian.

### Cây phân đoạn động

Gọi như vậy vì hình dạng của nó là động và các nút thường được phân bổ động. Còn được gọi là *cây phân đoạn ngầm* hoặc *cây phân đoạn thưa thớt* . Trước đây, chúng ta đã xem xét các trường hợp khi chúng ta có khả năng xây dựng cây phân đoạn ban đầu. Nhưng phải làm gì nếu kích thước ban đầu chứa đầy một số phần tử mặc định, nhưng kích thước của nó không cho phép ta xây dựng hoàn toàn trước nó?

Chúng ta có thể giải quyết vấn đề này bằng cách tạo cây phân đoạn một cách lười biếng (tăng dần). Ban đầu, chúng ta sẽ chỉ tạo gốc và sẽ chỉ tạo các đỉnh khác khi

cần. Trong trường hợp này, chúng ta sẽ sử dụng cách triển khai trên các con trỏ (trước khi đi đến các đỉnh con, hãy kiểm tra xem chúng đã được tạo chưa và nếu chưa thì hãy tạo chúng). Mỗi truy vấn vẫn chỉ có độ phức tạp O(logn) đủ nhỏ cho hầu hết các trường hợp sử dụng (ví dụ: log^2 10^9 ≃ 30)

Trong quá trình triển khai này, chúng ta có hai truy vấn, thêm một giá trị vào một vị trí (ban đầu tất cả các giá trị đều được 0) và tính tổng của tất cả các giá trị trong một phạm vi; vertex(0, n) sẽ là đỉnh gốc của cây ẩn.

Rõ ràng ý tưởng này có thể được mở rộng theo nhiều cách khác nhau. Ví dụ: bằng cách thêm hỗ trợ cho các cập nhật phạm vi thông qua việc lan truyền lười biếng.

## Phần II: BÀI TẬP VẬN DỤNG

**Bài toán 1:** [**http://www.spoj.com/problems/KQUERY/**](http://www.spoj.com/problems/KQUERY/)

### Tóm tắt đề:

Cho dãy số nguyên dương n phần tử a1, a2, a3, ...,an và q truy vấn. Mỗi truy vấn q có dạng bộ ba số i, j, k ( 1<= i <= j <= n). Với mỗi truy vấn trả về số phần tử lớn hơn k trong dãy ai, ai+1, …, aj. (1 <= n <= 3e4; 1 <= ai <= 1e9; 1 <= q <= 2e5)

### Gợi ý cách giải:

* + Với mỗi note trong cây segmenttree ta lưu trữ các phần tử được quản lý bởi note đó (đã được sắp xếp không giảm).
  + Ta tiến hành chặt nhị phân ở các note thuộc đoạn [i, j] để đếm số phần tử thỏa mãn thuộc note đó

**Bài toán 2:** [**https://codeforces.com/problemset/problem/339/D**](https://codeforces.com/problemset/problem/339/D)

### Tóm tắt đề:

Cho dãy số nguyên dương 2n phần tử a1, a2, a3, ...,an và q truy vấn cập nhật. Với mỗi truy vấn q có dạng p, q là thay thế phần tử ở vị trí p là q. Sau khi thực hiện mỗi truy vấn cần tính độ đẹp của dãy. *Độ đẹp được định nghĩa thông qua các bước như sau:* Chia dãy thành n cặp liên tiếp nhau để tính phép **or** giữa các cặp số và thu được 1 dãy 2(n-1) phần tử sau đó thực hiện phép tính phép **xor** giữa các cặp số liên tiếp nhau. Ta thực hiện lần lượt 2 thao tác này cho đến khi dãy chỉ còn 1 phần tử là v.

Ví dụ ta có dãy a = (1, 2, 3, 4). Các lần thực hiện sẽ là: (1, 2, 3, 4) → (1 or 2 = 3, 3 or 4

= 7) → (3 xor 7 = 4). Vậy độ đẹp của dãy trên là v = 4.

*Giới hạn:* 0 <= ai < 230, 1 <= n <= 17)

### Gợi ý cách giải:

Ta có số phần tử của mảng là 2n → cây có n+1 bậc với bậc thứ 0 là note 1. Mà ta thấy rằng phép toán or và xor được thực hiện luân phiên nhau nhưng or là phép toán được sử dụng đầu tiên → tại bậc thứ n+1 ta sẽ tính bằng phép toán or, bậc n bằng phép toán xor,...

**Bài toán 3:** [**https://www.spoj.com/problems/GSS3/**](https://www.spoj.com/problems/GSS3/)

### Tóm tắt đề:

Cho dãy số nguyên dương n phần tử a1, a2, a3,..., an và q truy vấn có dạng:

*Dạng 1:* 0 x y: cập nhật giá trị ở vị trí x thành y

*Dạng 2:* 1 x y: in ra dãy con liên tiếp có tổng lớn nhất nằm trong đoạn

x,y

(1 <= n <= 5e4; |ai| <= 1e4; 1 <= q <= 5e4)

### Gợi ý cách giải:

Ta sẽ lưu trữ 4 giá trị của mỗi đỉnh: tổng của phân đoạn, tổng tiền tố tối đa, tổng hậu tố tối đa, và tổng của phân đoạn con tối đa trong đó.

**Bài toán 4:** [**https://codeforces.com/problemset/problem/1234/D**](https://codeforces.com/problemset/problem/1234/D)

### Tóm tắt đề:

Cho 1 xâu chỉ chứa các ký tự thường có độ dài không quá 1e5 và q truy vấn có dạng:

*Dạng 1:* 1 pos c: thay thế ký tự ở vị trí pos bằng ký tự c

*Dạng 2:* 2 l r: in ra số lượng ký tự phân biệt từ l đến r với 1 <= n, q <= 1e5.

### Gợi ý cách giải:

Tại các note ta cần cho biết ký tự x (‘a’ <= x <= ‘z’) có xuất hiện hay không → ta sẽ lưu bằng mảng seg[4\*n][x] với chiều thứ hai cho biết note đó chứa bao nhiêu ký tự x; **Bài toán 5:** [**https://codeforces.com/problemset/problem/558/E**](https://codeforces.com/problemset/problem/558/E)

### Tóm tắt đề:

Cho 1 xâu chỉ chứa các ký tự thường có độ dài không quá 1e5 và q truy vấn. Mỗi truy vấn có dạng (i, j, k). Với k bằng 0 thì hãy sắp xếp các ký tự từ vị trí i đến j theo thứ tự giảm dần và với k bằng 1 thì sắp xếp theo thứ tự tăng dần. (1 <= q <= 5e5)

### Gợi ý cách giải:

Ta sử dụng kĩ thuật **counting sort**: trong đoạn i j ta cần biết được số lần xuất hiện của các ký tự từ ‘a’ đến ‘z’ sau đó dùng kĩ thuật lazy để có thể update 1 đoạn liên tiếp chứa các ký tự tùy theo đó là truy vấn sắp xếp tăng dần hay giảm dần.

**Bài toán 6:** [**https://codeforces.com/problemset/problem/920/F**](https://codeforces.com/problemset/problem/920/F)

### Tóm tắt đề:

Gọi D(x) là số lượng ước số của x. Ví dụ D(6) = 4 vì 1, 2, 3, 6 là ước số của 6. Cho dãy a gồm n phần tử nguyên dương a1, a2, ..., an và q truy vấn (1<= n, q <=3e5; 1 <= ai

<= 1e6). Mỗi truy vấn có dạng:

*Dạng 1:* 1 l r: với mọi i nằm trong đoạn [l, r] thay thế ai bằng D(ai)

*Dạng 2:* 2 l r: tính tổng các ai với l <= i <= r.

### Gợi ý cách giải:

Vì 1 <= ai <= 1e6 → ta dùng sàng ước để tính số ước của 1 số bất kì <= 1e6. Ta sẽ lưu 2 mảng là st[note] và maxn[note] cho biết tổng của note và giá trị lớn nhất của note đó.

Nhận xét rằng khi update, nếu giá trị lớn nhất của note đó > 2 thì trong dãy mà note đó quản lý vẫn tồn tại phần tử có thể gán bằng số ước của chính nó.

**Bài toán 7:** [**https://codeforces.com/problemset/problem/242/E**](https://codeforces.com/problemset/problem/242/E)

### Tóm tắt đề:

Cho dãy a gồm n phần tử nguyên dương a1, a2, ..., an và q truy vấn có dạng:

*Dạng 1:* 1 l r: tính tổng các phần tử ở vị trí l đến r

*Dạng 2:* 2 l r x: cập nhật các vị trí i thỏa mãn l <= i <= r thành ai = ai *xor* x (1 <= n <= 1e5; 1 <= m <= 5e4; 1 <= ai, xi <= 1e6)

### Gợi ý cách giải:

Ta thấy rằng 1 <= ai, xi <= 1e6 → các thao tác update luôn trả về các phần tử <= 1e6. Ta tạo mảng seg[note][20] để biểu diễn tổng của note dưới dạng bit (220 > 1e6)

.**Phần III: BÀI TẬP LÀM THÊM**

|  |  |
| --- | --- |
| **Cơ bản** | **Nâng cao** |
| <https://codeforces.com/contest/339/problem/D> <https://codeforces.com/contest/356/problem/A> <https://codeforces.com/contest/61/problem/E> <https://codeforces.com/contest/474/problem/F> <https://codeforces.com/contest/380/problem/C> <https://codeforces.com/contest/220/problem/E> <https://codeforces.com/contest/338/problem/E> <https://codeforces.com/contest/19/problem/D> <https://codeforces.com/contest/351/problem/D> <https://codeforces.com/contest/515/problem/E> <https://codeforces.com/contest/609/problem/F> <https://codeforces.com/contest/594/problem/D> <https://codeforces.com/contest/455/problem/E> | **Lazy Propagating:** <https://codeforces.com/contest/52/problem/C> <https://codeforces.com/contest/145/problem/E> <https://codeforces.com/contest/558/problem/E> <https://codeforces.com/contest/240/problem/F> <https://codeforces.com/contest/446/problem/C> <https://codeforces.com/contest/115/problem/E> <https://codeforces.com/contest/438/problem/D> <https://codeforces.com/contest/121/problem/E> <https://codeforces.com/contest/610/problem/E> <https://codeforces.com/contest/580/problem/E> Segment tree with Vector: <https://codeforces.com/contest/369/problem/E> <https://codeforces.com/contest/610/problem/D> Segment Tree & Dp: <https://codeforces.com/contest/474/problem/E> <https://codeforces.com/contest/597/problem/C>  https://codeforces.com/contest/56/problem/E |

## Phần V: TÀI LIỆU THAM KHẢO

1. Website: <https://cp-algorithms.com/data_structures/segment_tree.html>
2. Website: <https://vnoi.info/wiki/algo/data-structures/segment-tree-extend>
3. Website: <https://vn.spoj.com/>
4. Website: <https://codeforces.com/>
5. Website: <http://lqdoj.edu.vn/>
6. Website:https://leetcode.com