# Cây phân đoạn

**Cây phân đoạn** (Segment tree) là một cấu trúc dữ liệu cho phép trả lời các ***truy vấn trong một phạm vi liên tiếp trên một mảng*** một cách hiệu quả, trong khi vẫn đủ linh hoạt để ***cho phép sửa đổi mảng***. Cụ thể hơn, cây phân đoạn cho phép `tìm tổng các phần tử hoặc tìm phần tử nhỏ nhất (lớn nhất) trong một đoạn liên tiếp a[l... r]`, trong thời gian `O(logn)`. Bên cạnh việc trả lời các truy vấn như vậy, cây phân đoạn cũng cho phép sửa đổi mảng bằng cách `thay đổi một phần tử hoặc thậm chí thay đổi các phần tử của toàn bộ một đoạn liên tiếp` (ví dụ: gán tất cả các phần tử a[l... r] cho một giá trị x bất kì hoặc cộng một giá trị x vào tất cả các phần tử trong đoạn).

Nói chung, cây phân đoạn là một cấu trúc dữ liệu rất linh hoạt và một số lượng lớn các vấn đề có thể được giải quyết với nó. Ngoài ra, cũng có thể áp dụng các thao tác phức tạp hơn và trả lời các truy vấn phức tạp hơn (xem Phiên bản nâng cao của Cây phân đoạn). Đặc biệt, Cây Phân đoạn có thể dễ dàng khái quát hóa đến kích thước lớn hơn. Ví dụ: với Cây Phân đoạn hai chiều, bạn có thể trả lời tổng hoặc các truy vấn tối thiểu trên một số phần phụ của một ma trận đã cho. Tuy nhiên, độ phức tạp sẽ là `O(log2(n) ^ 2)`.

Một tính chất quan trọng là Segment Tree chỉ yêu cầu một lượng bộ nhớ tuyến tính. Cây Phân đoạn tiêu chuẩn yêu cầu bộ nhớ 4*n để làm việc trên một mảng kích thước n.

# Dạng đơn giản nhất của Cây Phân đoạn

Để bắt đầu, chúng ta hãy xem xét hình thức đơn giản nhất của Cây phân đoạn. Ở đây, chúng ta muốn trả lời các truy vấn tổng một cách hiệu quả. Yêu cầu cụ thể sẽ là : cho một mảng a[0... n - 1], ta cần xây dựng một Cây Phân đoạn có khả năng tìm tổng các phần tử trong một đoạn [l,r] (tức là tính tổng `∑a[i] (i=l->r)`), cũng như có thể thay đổi giá trị của các phần tử trong mảng (tức là thực hiện phép gán a[i]=x). Cây Phân đoạn sẽ có thể xử lý cả hai truy vấn trong thời gian `O(logn)`.

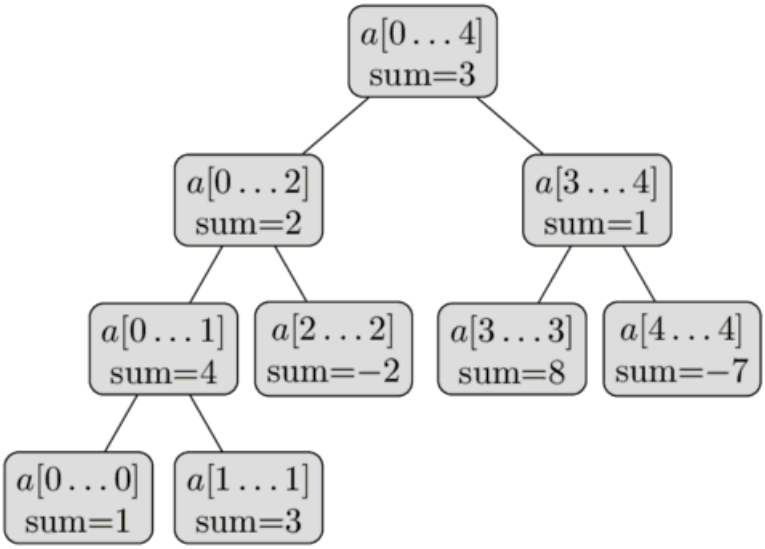
## Cấu trúc của Cây Phân đoạn

Vậy, Cây Phân đoạn là gì?

Chúng ta tính toán và lưu trữ tổng các phần tử của toàn bộ mảng, tức là tổng của phân đoạn a[0... n - 1]. Sau đó, chúng ta chia mảng thành hai nửa a[0... n/2] và a[n/2+1... n - 1] rồi tính tổng của mỗi nửa và lưu trữ chúng. Mỗi nửa trong hai nửa này lần lượt cũng chia làm đôi, tổng của chúng được tính toán và lưu trữ. Và quá trình này lặp lại cho đến khi tất cả các phân đoạn đạt kích thước 1. Nói cách khác, chúng ta ***bắt đầu với phân đoạn a[0... n - 1], chia đôi phân đoạn hiện tại (nếu nó chưa trở thành một phân đoạn chứa một phần tử duy nhất), sau đó tiếp tục thực hiện quy trình này cho cả hai nửa con của phân đoạn hiện tại.*** Đối với mỗi phân đoạn như vậy, chúng ta lưu trữ tổng toàn bộ các phần tử của phân đoạn đó.

Chúng ta có thể nói rằng các phân đoạn này tạo thành một cây nhị phân: gốc của cây này là phân đoạn a[0...n - 1] và mỗi nút (ngoại trừ nút lá) có chính xác hai nút con. Đây là lý do tại sao cấu trúc dữ liệu được gọi là "Cây phân đoạn", mặc dù trong hầu hết các cách cài đặt, cấu trúc cây không được xây dựng một cách rõ ràng (xem Thực hiện).

Dưới đây là một biểu diễn trực quan của Cây Phân đoạn như vậy trên mảng a=[1,3,−2,8,−7]:




Từ mô tả này, chúng ta đã có thể kết luận rằng *`Cây Phân đoạn chỉ yêu cầu một số lượng các nút tuyến tính`*. 'Tầng' đầu tiên của cây chứa một nút duy nhất (gốc), tầng thứ hai sẽ chứa hai nút, trong tầng thứ ba, nó sẽ chứa bốn nút, cho đến khi số nút đạt đến n. Do đó, số lượng nút trong trường hợp xấu nhất có thể được ước tính bằng tổng 1+2+4+⋯+2^⌈log2(n)⌉=2⌈log2(n)⌉+1 < 4n.

Điều đáng chú ý là bất cứ khi nào n không phải luỹ thừa của 2, thì không phải tầng nào của Cây Phân đoạn cũng sẽ được lấp đầy hoàn toàn. Chúng ta có thể thấy này trong hình minh hoạ trên. Bây giờ chúng ta có thể quên đi thực tế này, nhưng nó sẽ trở nên quan trọng sau này trong quá trình cài đặt.

`Chiều cao của Cây Phân đoạn là O(logn)`, bởi vì khi đi xuống từ gốc đến lá kích thước của các phân đoạn liên tục giảm đi khoảng một nửa.

## Xây dựng

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 một cây phân đoạn tính tổng tổng, một nút sẽ lưu trữ tổng các phần tử trong phạm vi [l,r] của nó.
2. thao tác `hợp nhất (merge)` hai nút 'anh em' trong một cây phân đoạn. Ví dụ: trong một 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 nút là "nút lá", nếu phân đoạn tương ứng của nó chỉ bao gồm một phần tử trong mảng ban đầu. Nó có mặt ở cấp dưới cùng của một 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 dưới cùng (nút lá) và gán cho chúng các giá trị tương ứng của chúng. Trên cơ sở các giá trị này, chúng ta có thể tính toán các giá trị của tầng trước đó, sử dụng hàm `merge`. Và trên cơ sở đó, chúng ta có thể tính toán các giá trị của tầng trước đó và lặp lại quy trình cho đến khi chúng ta đạt đến nút gốc.**

Một cách thuận tiện chúng ta có thể mô tả quá trình này theo hướng `đệ quy`, tức là `từ nút gốc đến nút lá`. Quá trình xây dựng đệ quy này, nếu `được gọi trên nút không phải nút lá`, thực hiện như sau:

1. Gọi đệ quy xây dựng các giá trị của hai nút con
2. Hợp nhất các giá trị tính toán của những nút con này để tạo ra giá trị cho nút hiện tại

Chúng ta bắt đầu xây dựng ở nút 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 độ phức tạp thời gian là hằng số (thao tác hợp nhất được gọi `O(n)` lần, bằng với số lượng nút bên trong trong cây phân đoạn).

## Truy vấn tính tổng

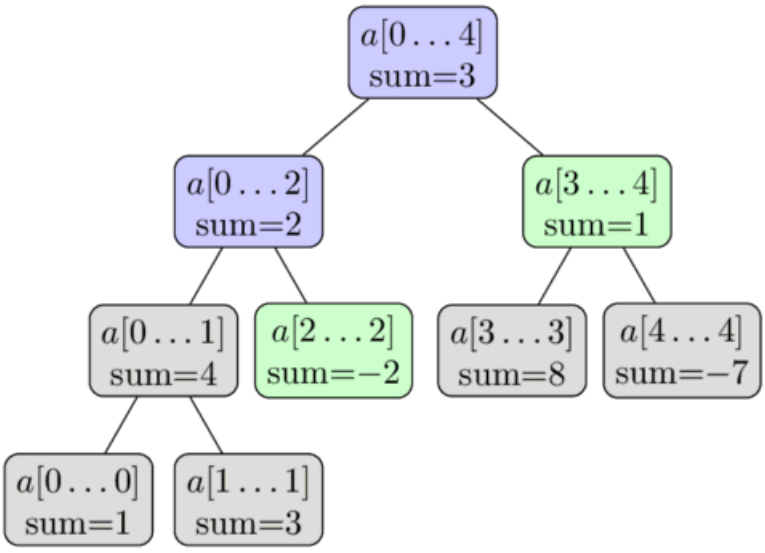
Bây giờ chúng ta sẽ trả lời các truy vấn tính tổng. Trong input cho 2 số nguyên l và r, chúng ta phải tính tổng của đoạn a[l... r] trong `O(logn)`.

Để làm điều này, chúng ta sẽ ***đi qua Cây Phân đoạn và sử dụng các tổng được tính trước (ở phần xây dựng) của các phân đoạn***. Giả sử rằng chúng ta hiện đang ở nút quản lí phân đoạn a[tl... tr] (bắt đầu thực hiện xử lí truy vấn từ nút gốc), ta muốn `tìm tổng phần giao của đoạn a[l..r] với nút này`. Có ba trường hợp có thể xảy ra:

1. Nút a[tl... tr] không giao với a[l..r] (tức là l > tr hoặc tl > r). Lúc này chúng ta *trả về 0*.
2. Nút a[tl... tr] nằm hoàn toàn trong a[l..r] (tức là l <= tl và tr <= r). Lúc này chúng ta lấy toàn bộ nút, *trả về tổng được tính trước và lưu trữ trong nút*.
3. Trường hợp cuối cùng, a[tl... tr] giao một phần với a[l..r] (tức không thoả mãn cả hai điều kiện trên). Gọi đệ quy cho hai nút con để tìm tổng phần giao giữa a[l..r] và từng nút con của nút hiện tại, *trả về tổng của hai giá trị này*.

Một cách ngắn gọn, quá trình truy vấn này tách đoạn [l...r] thành các đoạn [tl...tr] không giao nhau (trường hợp 2) rồi lấy tổng của chúng, như vậy sẽ lấy được tổng của đoạn [l..r]. Hai trường hợp còn lại có nhiệm vụ tách các nút để đi đến trường hợp 2.

Quá trình được minh họa qua hình dưới đây. Cho a=[1,3,−2,8,−7], và ở đây chúng ta muốn tính tổng a[2..4]. Các nút màu xanh lá sẽ là những nút được chọn để lấy tổng (rơi vào trường hợp 2). Điều này cho chúng ta kết quả −2+1=−1.




Độ phức tạp của truy vấn là `O(logn)`. Để làm rõ, chúng ta xét từng tầng của cây. Với mỗi tầng, chúng ta chỉ truy cập không quá bốn nút. 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 (tối đa bốn nút mỗi tầng) là đúng bằng quy nạp. - Ở tầng đầu tiên, chúng ta chỉ ghé thăm một nút gốc, vì vậy ở đây chúng ta ghé thăm ít hơn bốn nút. - Bây giờ chúng ta hãy xem xét một tầng tùy ý. Theo giả thuyết quy nạp, chúng ta ghé thăm nhiều nhất bốn nút. Nếu chúng ta chỉ ghé thăm tối đa hai nút, tầng tiếp theo có tối đa bốn nút (vì mỗi nút chỉ có tối đa hai nút con). Vì vậy, giả sử rằng chúng ta truy cập ba hoặc bốn nút ở tầng hiện tại. Từ những nút đó, chúng ta sẽ phân tích các nút ở giữa cẩn thận hơn. Vì truy vấn tổng yêu cầu tính tổng của một đoạn liên tiếp, chúng ta biết rằng các phân đoạn tương ứng với các nút được 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 cuộc gọi đệ quy nào. Vì vậy, chỉ còn lại nút bên trái nhất, và nút phải nhất sẽ có khả năng thực hiện các cuộc gọi đệ quy. Và những cuộc gọi đó sẽ chỉ tạo ra tối đa bốn cuộc gọi đệ quy, vì vậy cấp độ tiếp theo cũng sẽ đúng với khẳng định trên. Chúng ta có thể nói rằng một nhánh tiếp cận biên trái của truy vấn và nhánh thứ hai tiếp cận biên phải.
>
> Do đó, chúng ta truy cập nhiều nhất 4logn nút tổng cộng, và tương đương với thời gian chạy O(logn).

Tóm lại, truy vấn hoạt động bằng cách chia phân đoạn đầu vào thành nhiều phân đoạn con mà tổng các đoạn con đó đã được tính toán trước và lưu trữ trong cây. Và nếu chúng ta ngừng chia bất cứ khi nào phân đoạn truy vấn bao trùm phân đoạn của nút, thì chúng ta chỉ cần O(logn) các phân đoạn con như vậy, điều đem lại hiệu quả cho Cây Phân đoạn.

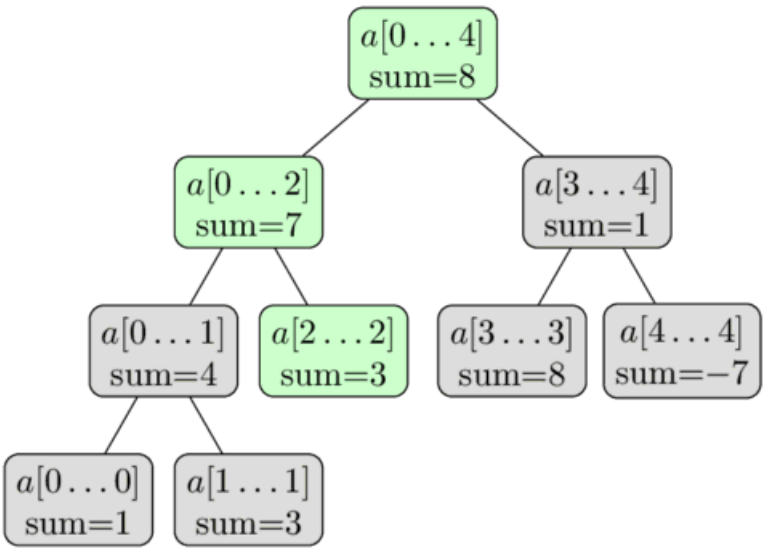
## Truy vấn Cập nhật

Bây giờ chúng ta muốn thay đổi một phần tử cụ thể trong mảng, giả sử chúng ta muốn thực hiện phép gán a[i]=x. Và chúng ta phải xây dựng lại Segment Tree, sao cho nó tương ứng với mảng mới vừa được sửa đổi.

Truy vấn này dễ dàng hơn truy vấn tổng. Mỗi tầng của Cây Phân đoạn tạo thành một cách chia của mảng. Một phần tử a[i] chỉ đóng góp vào (nằm trong) một nút tại mỗi tầng. Như vậy khi thay đổi một phần tử a[i] thì chỉ có tối đa O(logn) nút cần được cập nhật.

Thật dễ dàng để thấy rằng *yêu cầu cập nhật có thể được thực hiện bằng cách sử dụng hàm đệ quy* . Hàm này được gọi từ nút gốc, `mỗi lần từ nút hiện tại nó sẽ tự gọi đệ quy tới một trong hai nút con (nút con chứa a[i] trong đoạn mà nó quản lí), và sau đó tính toán lại giá trị tổng của nó`, tương tự như cách nó được thực hiện trong phần xây dựng cây (đó là tổng hai nút con của nó).

Một lần nữa sau đây là hình minh hoạ truy vấn cập nhật sử dụng cùng một mảng như trên. Ở đây chúng ta thực hiện cập nhật a[2]=3. Các nút màu xanh lá cây là các nút mà chúng ta gọi đệ quy đến và cập nhật.




## Cài đặt

Điều đáng cân nhắc là làm thế nào để lưu trữ Cây phân đoạn. Tất nhiên chúng ta có thể định nghĩa một cấu trúc nút và tạo các đối tượng, lưu trữ ranh giới [l,r] của phân đoạn, tổng của nó và ngoài ra cũng trỏ đến các đỉnh con của nó. Tuy nhiên, điều này đòi hỏi phải lưu trữ rất nhiều thông tin dư thừa. Chúng ta sẽ sử dụng một mẹo đơn giản, để làm cho việc này hiệu quả hơn rất nhiều. Chúng ta chỉ **lưu trữ các giá trị của các nút trong một mảng**. Nút gốc có chỉ số 1, hai nút con của nút gốc có chỉ số 2 và 3, lần lượt hai nút con của hai nút này nằm ở các chỉ số 4 đến 7, v.v. Thật dễ dàng để thấy rằng nút con bên trái của một nút có chỉ số `i` thì có chỉ số `2i` và nút con bên phải là `2i + 1` .

Điều này đơn giản hóa việc cài đặt rất nhiều. Chúng ta không cần phải lưu trữ cấu trúc của cây trong bộ nhớ. Nó được định nghĩa ngầm. *Chúng ta chỉ cần một mảng chứa tổng của tất cả các phân đoạn mà các nút quản lí.*

Như đã lưu ý trước đây, chúng ta cần lưu trữ nhiều nhất 4n nút. Nó có thể ít hơn, nhưng để thuận tiện, chúng ta `luôn khai báo một mảng kích thước 4n` . Sẽ có một số phần tử dư thừa trong mảng không tương ứng với bất kỳ nút nào trong cây, nhưng điều này không làm phức tạp việc cài đặt.

Vì vậy, chúng ta lưu trữ Segment Tree trong một mảng t[] với kích thước 4n:

```
type intArr = array[1..4*MAXN] of longint;
var n : longint;
    a,t : intArr;
```

Quy trình xây dựng Cây Phân đoạn từ một mảng a[] là một hàm đệ quy với các tham số: a[] (mảng đầu vào), v (chỉ số của nút hiện tại) và ranh giới [tl...tr] mà nút v quản lí. Trong chương trình chính, hàm này sẽ được gọi với các tham số của nút gốc: v=1, tl=0và tr=n−1.

```
procedure build(var a : intArr; v,tl,tr : longint);
    var tmid : longint;
    begin
        if(tl = tr) then t[v] := a[tl]
        else begin
            tmid := (tl + tr) div 2;
            build(a,v * 2, tl,tmid);
            build(a,v * 2 + 1,tmid + 1,tr);
            t[v] := t[v * 2] + t[v * 2 + 1];
        end;
    end;
```

Ngoài ra, hàm trả lời các truy vấn tổng cũng là một hàm đệ quy, nhận tham số về nút hiện tại (tức là chỉ số v và ranh giới [tl...tr]) và cả thông tin về ranh giới [l...r] của truy vấn. Để đơn giản, hàm này luôn thực hiện hai cuộc gọi đệ quy, ngay cả khi chỉ cần một cuộc gọi - trong trường hợp đó, cuộc gọi đệ quy thừa sẽ có l > r và trường hợp này có thể dễ dàng loại bỏ bằng cách sử dụng một điều kiện kiểm tra ở đầu hàm.

```
function sum(v, tl, tr, l, r : longint):longint; // trả về tổng của phần giao của [tl,tr] và [l,r]
    var tmid : longint;
    begin
        if (l > r) or (l > tr) or (tl > r) then exit(0); // nếu 2 đoạn không giao nhau thì exit
        if(l <= tl) and (tr <= r) then exit(t[v]); // nếu [tl,tr] nằm hoàn toàn trong [l,r] thì lấy cả nút này
        tmid := (tl + tr) div 2;
        exit(sum(v * 2, tl, tmid, l, r) + sum(v * 2 + 1, tmid + 1, tr, l, r)); // gọi truy vấn tới 2 nút con rồi tính tổng
    end;
```

Cuối cùng là truy vấn cập nhật. Hàm cập nhật sẽ nhận được thông tin về nút hiện tại và ngoài ra còn có thông tin về truy vấn cập nhật (tức là vị trí của phần tử và giá trị mới của nó).

```
procedure update(v, tl, tr, pos, new_val : longint); // a[pos] = new_val
    var tmid : longint;
    begin
        if(tl == tr) then t[v] := new_val;
        else begin
            tmid := (tl + tr) div 2;
            if(pos <= tmid) then update(v * 2, tl, tmid, pos, new_val)
            else update(v * 2 + 1,tmid + 1,tr,pos,new_val);
            t[v] := t[v * 2] + t[v * 2 + 1]; // cập nhật lại tổng của nút v
        end;
    end;
```

### Triển khai hiệu quả bộ nhớ

Bộ nhớ được sử dụng theo phương pháp cài đặt trên là 4n, tuy vậy Segment Tree chỉ thực sự có khoảng 2n - 1 nút. Có một số cách có thể đánh số lại các nút để giảm bộ nhớ sử dụng nhưng trong những trường hợp thông thường thì việc giảm bộ nhớ này cũng không thực sự cần thiết.

## Những phiên bản nâng cao của Segment Tree

Cây phân đoạn là một cấu trúc dữ liệu rất linh hoạt, cho phép điều chỉnh tạo ra các biến thể và mở rộng theo nhiều hướng khác nhau. Có một số loại phổ biến như dưới đây.

### Truy vấn phức tạp hơn

Có thể khá dễ dàng để thay đổi Cây phân đoạn sao cho nó tính toán các truy vấn khác nhau (ví dụ: tính toán min / max trong một đoạn thay vì tổng), nhưng nó cũng có thể không hoàn toàn đơn giản.

#### Tìm giá trị lớn nhất

Hãy thay đổi điều kiện của bài toán được mô tả ở trên một chút: thay vì truy vấn tổng, bây giờ chúng ta sẽ thực hiện các truy vấn tìm giá trị lớn nhất của một đoạn.

Cây sẽ có cấu trúc giống hệt như cây được mô tả ở trên. Chúng ta chỉ cần thay đổi cách thức t[v] được tính toán trong hàm build và hàm update. t[v] bây giờ sẽ lưu trữ giá trị max của đoạn mà nó quản lí. Và chúng ta cũng cần thay đổi cách tính toán giá trị trả về của truy vấn (thay thế tổng bằng giá trị max).

Tất nhiên vấn đề này có thể dễ dàng thay đổi thành tính min thay vì tính max.

Phần cài đặt cho sẽ được đưa ra ở một phiên bản phức tạp hơn của bài toán này trong phần tiếp theo.

#### Tìm giá trị lớn nhất và số lần nó xuất hiện trong đoạn

Bài toán này rất giống với bài toán trước đó. Ngoài việc tìm kiếm giá trị max, chúng ta cũng phải tìm số lần xuất hiện của nó.

Để giải quyết bài toán này, chúng ta *lưu trữ một cặp số ở mỗi nút trong cây (t[v]).* Ngoài giá trị max của nút, chúng ta cũng lưu trữ số lần xuất hiện của nó trong đoạn mà nút này quản lí. t[v] vẫn có thể được tính trong thời gian hằng số bằng cách sử dụng thông tin của các cặp giá trị được lưu trữ ở nút con. Việc 'hợp nhất' hai cặp giá trị như vậy nên được cài đặt trong một hàm riêng biệt, vì đây sẽ là một hoạt động mà chúng ta sẽ thực hiện trong khi xây dựng cây, đồng thời khi trả lời các truy vấn lấy max cũng như trong khi thực hiện các thao tác cập nhật.

```
pair<int, int> t[4*MAXN];

pair<int, int> combine(pair<int, int> a, pair<int, int> b) {
    if (a.first > b.first)
        return a;
    if (b.first > a.first)
        return b;
    return make_pair(a.first, a.second + b.second);
}

void build(int a[], int v, int tl, int tr) {
    if (tl == tr) {
        t[v] = make_pair(a[tl], 1);
    } else {
        int tm = (tl + tr) / 2;
        build(a, v*2, tl, tm);
        build(a, v*2+1, tm+1, tr);
        t[v] = combine(t[v*2], t[v*2+1]);
    }
}

pair<int, int> get_max(int v, int tl, int tr, int l, int r) {
    if (l > r)
        return make_pair(-INF, 0);
    if (l == tl && r == tr)
        return t[v];
    int tm = (tl + tr) / 2;
    return combine(get_max(v*2, tl, tm, l, min(r, tm)),
                   get_max(v*2+1, tm+1, tr, max(l, tm+1), r));
}

void update(int v, int tl, int tr, int pos, int new_val) {
    if (tl == tr) {
        t[v] = make_pair(new_val, 1);
    } else {
        int tm = (tl + tr) / 2;
        if (pos <= tm)
            update(v*2, tl, tm, pos, new_val);
        else
            update(v*2+1, tm+1, tr, pos, new_val);
        t[v] = combine(t[v*2], t[v*2+1]);
    }
}
```

### Tính toán ước chung lớn nhất (GCD) / bội chung nhỏ nhất (LCM) của một đoạn

Trong vấn đề này, chúng ta muốn tính toán GCD / LCM của các đoạn trong mảng.

Biến thể thú vị này của Cây Phân đoạn có thể được giải quyết chính xác theo cách giống như Cây Phân đoạn mà chúng ta thực hiện các truy vấn tính tổng / min / max: lưu trữ GCD / LCM của đỉnh tương ứng trong mỗi đỉnh của cây. GCD của nút hiện tại là GCD của hai nút con, tương tự với LCM.

### Đếm số lượng số 0, tìm kiếm số 0 thứ k

Trong vấn đề này, chúng ta muốn tìm số lượng số 0 trong một phạm vi nhất định và tìm vị trí của số 0 thứ k (từ trái sang phải) trong mảng a[].

Một lần nữa chúng ta phải thay đổi giá trị lưu trữ của cây một chút: Lần này chúng ta sẽ lưu trữ số lượng số 0 trong phân đoạn của mỗi nút vào t[]. Nó khá rõ ràng, cách xây dựng cây, cập nhật và hàm count_zero để đếm số lượng số 0 , chúng ta chỉ cần sử dụng các ý tưởng từ bài toán truy vấn tính tổng. Do đó, chúng ta đã giải quyết được phần đầu tiên của vấn đề này.

Bây giờ chúng ta phải tìm ra làm thế nào để giải quyết bài toán tìm kiếm số 0 thứ k trong mảng a[]. Để thực hiện truy vấn này, chúng ta sẽ bắt đầu từ đỉnh gốc và di chuyển mỗi lần sang nút con bên trái hoặc bên phải, tùy thuộc vào việc nút con nào chứ số 0 thứ k. Để quyết định chúng ta cần đi đến nút nào, chỉ cần xét các số 0 xuất hiện ở nút con bên trái là đủ. Nếu số lượng số 0 của nút con bên trái lớn hơn hoặc bằng k, thì ta sẽ tìm trong nút con bên trái, ngược lại thì ta sẽ tìm trong nút con bên phải. Lưu ý, nếu chúng ta chọn nút con bên phải, chúng ta phải trừ k cho số lượng số 0 của nút con bên trái.

Trong quá trình thực hiện, chúng ta có thể xử lý trường hợp đặc biệt a[] chứa ít hơn k số 0 bằng cách trả về -1.

```
int find_kth(int v, int tl, int tr, int k) {
    if (k > t[v])
        return -1;
    if (tl == tr)
        return tl;
    int tm = (tl + tr) / 2;
    if (t[v*2] >= k)
        return find_kth(v*2, tl, tm, k);
    else
        return find_kth(v*2+1, tm+1, tr, k - t[v*2]);
}
```

### Tìm kiếm tiền tố của mảng

Bài toán cụ thể như sau: cho một giá trị nhất định x, chúng ta phải tìm ra chỉ số `i` nhỏ nhất mà tổng của `i` phần tử đầu tiên 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 trong đoạn liên tiếp [0...i] với Cây phân đoạn. Tuy nhiên, điều này sẽ dẫn đến giải pháp O(logn^2).

Thay vào đó, chúng ta có thể sử dụng cùng một ý tưởng như trong phần trước (bài toán tìm số 0 thứ k) và tìm vị trí bằng cách hạ cây xuống: bằng cách di chuyển mỗi lần sang trái hoặc phải, tùy thuộc vào tổng của nút con bên trái. Do đó, kết quả được tìm ra trong thời gian O(logn).

### Tìm phần tử đầu tiên trong mảng lớn hơn x

Nhiệm vụ như sau: Với một giá trị nhất định x và một đoạn a[l... r], tìm i nhỏ nhất (đầu tiên) trong đoạn [l... r] mà a[i] lớn hơn x.

Bài toán này có thể được giải quyết bằng cách sử dụng tìm kiếm nhị phân, mỗi lần tính max của một đoạn liên tiếp [0...i] để so sánh với x bằng Cây phân đoạn. Tuy nhiên, điều này sẽ dẫn đến giải pháp O(logn^2).

Thay vào đó, chúng ta có thể sử dụng ý tưởng tương tự như trong các phần trước và tìm vị trí bằng cách hạ dần cây xuống: xuất phát từ nút gốc, mỗi lần di chuyển sang trái hoặc phải, tùy thuộc vào giá trị max của nút con bên trái. Do đó, kết quả được tìm ra trong thời gian O(logn).

```
int get_first(int v, int lv, int rv, int l, int r, int x) {
    if(lv > r || rv < l) return -1;
    if(l <= lv && rv <= r) {
        if(t[v] <= x) return -1;
        while(lv != rv) {
            int mid = lv + (rv-lv)/2;
            if(t[2*v] > x) {
                v = 2*v;
                rv = mid;
            }else {
                v = 2*v+1;
                lv = mid+1;
            }
        }
        return lv;
    }

    int mid = lv + (rv-lv)/2;
    int rs = get_first(2*v, lv, mid, l, r, x);
    if(rs != -1) return rs;
    return get_first(2*v+1, mid+1, rv, l ,r, x);
}
```

Ngoài ra còn một số bài toán nâng cao khác như :

### Finding subsegments with the maximal sum

### Saving the entire subarrays in each vertex

### Find the smallest number greater or equal to a specified number. No modification queries.

### Find the smallest number greater or equal to a specified number. With modification queries.

### Find the smallest number greater or equal to a specified number. Acceleration with "fractional cascading".

Storing data structures (Fenwick Trees, Cartesian trees, ...) in each vertex.

...

## Cập nhật một đoạn liên tiếp (Lazy Propagation)

Tất cả các bài toán trong các phần đã thảo luận trên chỉ nói về các truy vấn cập nhật một phần tử duy nhất của mảng. Tuy nhiên, Cây Phân đoạn cho phép áp dụng các truy vấn **thay đổi cho toàn bộ một đoạn các phần tử liền kề** và thực hiện truy vấn kết quả trong thời gian O(logn).

### Cộng vào một đoạn

Chúng ta bắt đầu bằng cách xem xét bài toán ở dạng đơn giản nhất: truy vấn cộng giá trị x vào tất cả các phần tử trong đoạn a[l... r]. Truy vấn thứ hai, yêu cầu trả về giá trị của a[i].

Để truy vấn cộng được hiệu quả, chúng ta lưu trữ ở mỗi nút trong Cây Phân đoạn lượng giá trị mà chúng ta phải cộng vào tất cả các phần tử mà nút đó quản lí . Ví dụ: nếu truy vấn "cộng 3 vào toàn bộ mảng a[0... n−1]" xuất hiện, thì chúng ta lưu giá trị 3 tại nút gốc. Nhìn chung, chúng ta phải đặt giá trị này vào nhiều nút được tách ra bởi đoạn [l,r]. Vì vậy, chúng ta không phải thay đổi tất cả O(n) giá trị, mà chỉ thay đổi O(logn) nút.

Nếu bây giờ có một truy vấn hỏi giá trị hiện tại của một phần tử a[i] cụ thể, chỉ cần đi trên cây từ trên xuống dưới và cộng tất cả các giá trị được tìm thấy trên đường đi là đủ.

```
procedure build(var a : intArr; v, tl, tr : longint);
    var tmid : longint;
    begin
        if(tl = tr) then t[v] := a[tl];
        else begin
            tmid := (tl + tr) div 2;
            build(a, v * 2, tl, tmid);
            build(a, v * 2 + 1, tmid + 1, tr);
            t[v] := 0;
        end;
    end;

procedure update(v, tl, tr, l, r, x : longint); // cộng x vào đoạn [l,r]
    var tmid : longint;
    begin
        if(l > r) or (l > tr) or (tl > r) then exit; // không giao nhau
        if (l <= tl) and (tr <= r) then t[v] := t[v] + x; // nếu [tl,tr] nằm trong [l,r]
        else begin
            tmid := (tl + tr) div 2;
            update(v * 2, tl, tmid, l, r, x); //
            update(v * 2 + 1, tmid + 1, tr, l, r, x); //
        end;
    end;

function get(v, tl, tr, pos : longint): longint; // lấy giá trị của a[pos] ở thời điểm hiện tại
    var tmid : longint;
    begin
        if(tl == tr) then exit(t[v]);
        tmid := (tl + tr) div 2;
        if(pos <= tmid) then exit(get(v * 2,tl,tmid,pos) + t[v]); // nếu pos nằm trong nút con bên trái thì gọi đệ quy sang nút bên
        else exit(get(v * 2 + 1,tmid + 1,tr,pos) + t[v]); // tương tự với bên phải
    end;
```

### Gán một đoạn

Giả sử bây giờ truy vấn cập nhật yêu cầu gán từng phần tử của một đoạn a[l... r] thành số p. Truy vấn thứ hai yêu cầu trả về giá trị của a[i].

Để thực hiện truy vấn cập nhật này trên toàn bộ đoạn, ta phải lưu trữ ở mỗi nút của Cây Phân đoạn một thông tin: `liệu phân đoạn mà nút này quản lí có được bao phủ bởi một phép gán bằng p nào hay không` . Điều này cho phép chúng ta thực hiện thao tác cập nhật "lazy (lười biếng)": trong mỗi truy vấn gán đoạn [l...r] bằng p, thay vì thay đổi tất cả phần tử trong đoạn, chúng ta chỉ thay đổi một số nút được tách ra từ đoạn [l...r] => thông tin của truy vấn sẽ được lưu vào các nút này.

***Theo một nghĩa nào đó, chúng ta 'lười biếng' và 'trì hoãn' việc gán giá trị mới cho các phần tử trong đoạn, thay vào đó truy vấn sẽ được lưu tạm vào các nút. Sau này, khi thực sự cần thiết, chúng ta mới lấy những truy vấn được lưu đó ra để đem gán cho các phần tử.***

Vì vậy, sau khi truy vấn cập nhật được thực hiện, một số phần của cây trở nên không liên quan - vì có thể việc gán vẫn chưa được thực hiện nên giá trị của chúng vẫn là giá trị cũ và không thực sự đúng.

Ví dụ: nếu truy vấn "gán p = 5 cho toàn bộ mảng a[0... n−1]" được thực hiện, thì trong Cây Phân đoạn chỉ có một thay đổi duy nhất - chính là giá trị p = 5 được đặt trong gốc của cây. Các phân đoạn còn lại vẫn không thay đổi, mặc dù trên thực tế, số p nên được đặt trong toàn bộ cây.

Giả sử bây giờ truy vấn gán thứ hai nói rằng nửa đầu của mảng a[0... n/2] nên được gán với một số p = 7. Để xử lý truy vấn này, chúng ta lẽ ra phải gán từng phần tử trong toàn bộ nút con bên trái của nút gốc với số p = 7 đó. Nhưng trước khi chúng ta làm điều này, trước tiên chúng ta phải giải quyết nút gốc trước. Sự tinh tế ở đây chính là nửa bên phải của mảng vẫn phải được gán cho giá trị của truy vấn đầu tiên (p = 5) và hiện tại không có thông tin cho nửa bên phải được lưu trữ.

Cách để giải quyết vấn đề này là 'đẩy' thông tin của nút gốc cho các nút con của nó, tức là nếu nút gốc được gán với bất kỳ số `p'` nào, thì chúng ta gán các nút con bên trái và bên phải với số `p'` này và loại bỏ số `p'` khỏi nút gốc (có thể đánh dấu lại hoặc gán bằng oo). Sau đó, chúng ta có thể gán nút bên trái với giá trị `p` mới mà không mất bất kỳ thông tin cần thiết nào.

Tóm lại: đối với bất kỳ truy vấn nào (truy vấn cập nhật hoặc lấy giá trị) **trong quá trình đi xuống dọc theo cây**, chúng ta **phải luôn đẩy thông tin từ nút hiện tại vào cả hai nút con của nó**. Chúng ta có thể hiểu điều này theo cách này: khi chúng ta đi xuống dưới, chúng ta *thực hiện các phép gán bị trì hoãn, nhưng chỉ ở mức cần thiết (để không làm suy giảm độ phức tạp truy vấn - O (logn))*.

Để thực hiện, chúng ta cần tạo một hàm push (đẩy), hàm này sẽ nhận tham số là nút hiện tại và nó sẽ đẩy thông tin trong nút này tới cả hai nút con của nó. Chúng ta sẽ gọi hàm này ở phần đầu của các hàm truy vấn (nhưng chúng ta sẽ không gọi nó từ các nút lá, vì nút lá không có nút con nên không cần đẩy nữa).

```
// marked[v] = đánh dấu liệu đang có phép gán nào cho toàn bộ nút v (được lưu tại t[v]) hay không

procedure push(v,tl,tr : longint);
    begin
        if(marked[v]) then begin
            if(tl < tr) then begin
                t[v * 2] := t[v];
                t[v * 2 + 1] := t[v];
                marked[v * 2] := true;
                marked[v * 2 + 1] := true;
            end;
            marked[v] := false;
        end;
    end;

procedure update(v, tl, tr, l, r, x : longint); // gán x cho đoạn [l,r]
    var tmid : longint;
    begin
        push(v, tl, tr); // đẩy giá trị được gán cho đoạn [tl,tr] xuống 2 nút con
        if (l > tr) or (tl > r) then exit; // không giao nhau
        if (l <= tl) and (tr <= r) then begin // nếu [tl,tr] nằm trong [l,r]
            t[v] := x;
            marked[v] := true;
            push(v,tl,tr); // tiếp tục đẩy xuống
        end else begin
            tmid := (tl + tr) div 2;
            update(v * 2, tl, tmid, l, r, x); //
            update(v * 2 + 1, tmid + 1, tr, l, r, x); //
        end;
    end;

function get(v, tl, tr, pos : longint): longint; // lấy giá trị của a[pos] ở thời điểm hiện tại
    var tmid : longint;
    begin
        push(v,tl,tr); // đẩy phép gán xuống 2 nút con
        if(tl == tr) then exit(t[v]);
        tmid := (tl + tr) div 2;
        if(pos <= tmid) then exit(get(v * 2,tl,tmid,pos)); // nếu pos nằm trong nút con bên trái thì gọi đệ quy sang nút bên trái
        else exit(get(v * 2 + 1,tmid + 1,tr,pos)); // tương tự với bên phải
    end;
```

### Cộng vào một đoạn, truy vấn giá trị max trong một đoạn

Bây giờ truy vấn cập nhật là cộng một số x vào tất cả các phần tử trong một đoạn và truy vấn đọc là tìm giá trị lớn nhất trong một đoạn.

Vì vậy, đối với mỗi nút của Cây phân đoạn, chúng ta phải lưu trữ giá trị max của phân đoạn tương ứng. Phần thú vị là làm thế nào để tính toán lại các giá trị này sau một yêu cầu cập nhật.

Vì mục đích này, chúng ta lưu trữ một giá trị bổ sung cho mỗi nút. Trong giá trị này, chúng ta lưu trữ tổng các giá trị X lẽ ra phải được cộng vào đoạn nhưng do 'trì hoãn' nên chúng ta chưa truyền đến các nút con. Trước khi đi đến nút con, chúng ta gọi hàm push để đẩy giá trị X đến cả hai nút con. Chúng ta phải làm điều này trong cả hàm cập nhật và hàm truy vấn.

```
// lazy[v] = tổng giá trị X phải được cộng vào nút v nhưng đang bị 'trì hoãn'
// t[v] = giá trị lớn nhất trong nút v

procedure push(v,tl,tr : longint);
    begin
        if(lazy[v] <> 0) then begin
            t[v] := t[v] + lazy[v]; // cập nhật giá trị t[v]
            if(tl < tr) then begin // đẩy tổng lazy[v] xuống
                lazy[v * 2] := lazy[v * 2] + lazy[v];
                lazy[v * 2 + 1] := lazy[v * 2 + 1] + lazy[v];
            end;
            lazy[v] := 0; // xoá lazy[v] vì lazy[v] đã được cập nhật vào t[v] rồi
        end;
    end;

procedure update(v, tl, tr, l, r, x : longint); // cộng x vào đoạn [l,r]
    var tmid : longint;
    begin
        push(v, tl, tr); // đẩy giá trị được cộng vào đoạn [tl,tr] xuống 2 nút con
        if (l > tr) or (tl > r) then exit; // không giao nhau
        if (l <= tl) and (tr <= r) then begin // nếu [tl,tr] nằm trong [l,r]
            lazy[v] := lazy[v] + x;
            push(v,tl,tr); // tiếp tục cập nhật t[v] và đẩy lazy[v] xuống
        end else begin
            tmid := (tl + tr) div 2;
            update(v * 2, tl, tmid, l, r, x); //
            update(v * 2 + 1, tmid + 1, tr, l, r, x); //
            t[v] := max(t[v * 2],t[v * 2 + 1]); // cập nhật lại giá trị lớn nhất trong nút v
        end;
    end;

function query(v, tl, tr, l, r : longint): longint; // lấy giá trị lớn nhất trong đoạn [l...r]
    var tmid : longint;
    begin
        push(v,tl,tr); // đẩy phép gán xuống 2 nút con
        if (l > tr) or (tl > r) then exit(0); // không giao nhau
        if (l <= tl) and (tr <= r) then exit(t[v]); // nếu [tl,tr] nằm trong [l,r]
        tmid := (tl + tr) div 2;
        exit(max(query(v * 2,tl, tmid,l,r),query(v * 2 + 1,tmid + 1,tr,l,r)));
    end;
```

## Bài tập

- SPOJ - KQUERY [Cây phân đoạn liên tục / Cây sắp xếp hợp nhất]
- Codeforces - Hoạt động xenia và bit
- UVA 11402 - Ahoy, Cướp biển!
- SPOJ - GSS3
- Codeforces - Truy vấn ký tự riêng biệt
- Codeforces - Knight Tournament [Dành cho người mới bắt đầu]
- Codeforces - Thuộc địa kiến
- Codeforces - Drazil và Park
- Codeforces - Thông tư RMQ
- Codeforces - Mảng may mắn
- Codeforces - Đứa trẻ và trình tự
- [Codeforces - DZY Yêu số Fibonacci Lazy propagation]
- Codeforces - Hoán vị bảng chữ cái
- Codeforces - Nhắm mắt
- Codeforces - Kefa và Watch
- Codeforces - Một nhiệm vụ đơn giản
- Codeforces - SUM và REPLACE
- [COCI - Deda Phần tử cuối cùng nhỏ hơn hoặc bằng x / Tìm kiếm nhị phân]
- Codeforces - Thời cổ đại không giám sát [2D]
- CSES - Truy vấn khách sạn
- CSES - Truy vấn đa thức
- CSES - Cập nhật phạm vi và tổng