Segment Tree

Segment tree - temel olarak bir dizi üzerinden girdi olarak sorgulanan l ve r değerlerinin arasındaki minimum ya da maksimum değeri sorgulayabilen hızlı bir veri yapısıdır. Bunun yanında dizinin elemanlarının ya da dizinin belli bir aralığındaki değişimleri de mümkün kılmaktadır(Örneğin bir aralıktaki tüm veriler bir sayıya eşitlenebilir yahut bunlara bir sayı eklenebilir).

Genel olarak segment tree esnek bir veri yapısıdır ve teorik olarak bir çok problemin çözümünde kullanılmıştır. Bunun yanında bir çok karmaşık işlem için de kullanılmaktadır ve bunlardan karmaşık segment tree versiyonları bölümünde anlatılacaktır. Ayrıca segment tree kolay olarak çok boyuta da adapte olabilmektedir.

Segment tree'nin önemli bir özelliği de linear hafıza kullanmasıdır. Standart segment tree 4n hafıza gerektirmektedir.

Standart Segment Tree'nin Açıklaması

İlk olarak segment tree'nin en kolay halini ele alalım - toplamlar için segment tree. Bir A dizisi üzerinde l ve r indisleri arasında kalan elemanların toplamını bulan ve aynı zamanda bir indisin yeni değerinin atanmasına (A[i]=x) imkan veren bir veri yapısı oluşturalım. Her iki işlemi de O(log n) zamanda gerçekleştirecektir.

Segment Tree'ye Giriş

Öncelikle şunu düşünelim bir aralığın toplamı aynı zamanda o aralığın ilk ve ikinci yarısının toplamına eşittir. Bizim çıkış noktamız da bunun üzerinedir : başlangıçta elimizde bulunan (0,n-1) aralığını iki parçaya ayırarak ilerleyeceğiz ta ki elimizde kalan aralık birim olana dek.

Şimdi şunu söyleyebiliriz ki bu oluşan tüm aralıklar içinden (0,n-1) aralığı bizim ağacımızın rootunu temsil etmektedir. Ayrıca ağaç üzerindeki her bir node(yaprak node'lar hariç yani birim uzunluklar) 2 çocuğa sahiptir ve çocukları sırasıyla ilk yarısı ve ikinci yarısıdır. Öte yandan ismi segment tree olsa ve mantığında ağaçları barındırsa dahi kodlanmasında baştan bir tree oluşturmayacağız.

Oluşan yapının linear büyüklükte olduğu görülmektedir ancak açıklanacak olursa ağacın ilk satırında yani root node'unda (0,n-1) aralığı vardır yani 1 elemandan oluşmaktadır. Ağaçta root'tan yapraklara doğru inildikçe eleman sayısı 2 katına çıkmaktadır ve en altta n eleman mevcuttur. Son duruma matematiksel olarak bakacak olursak $n + n/2 + n/4 + n/8 + \ldots. 1 < 2n$ dir.

Tree'nin yüksekliği: O(log n) her seviye inildiğinde aralıklar ikiye bölünerek ilerlenmektedir taki birim aralık olana dek. Bu da demektir ki bir aralık ancak log n defa bölünebilir.

Oluşturma

Oluşturma işlemi şu şekilde hızlı olarak gerçekleştirilebilir: Alttan üste doğru; öncelikle A[i] değerlerini yaprak node'lar a yazalım ardından bir üst seviyedeki node'ların değerleri çocuğu olarak sakladığı iki node'da bulunan değerlerin toplamı olacak şekilde güncellenir ve bu süreç root node'un değeri bulana dek devam eder.
Beklenen çalışma zamanı: O(n)

Sorgulama

Verilecek olan bir l ve r değeri için l ve r indisleri arasında kalan elemanların toplamlarını bulmamız gerekmektedir.

Bunu yapmak için root node'dan başlayarak istenilen aralığın toplamını bulmaya çalışacağız bu sırada bu node'un çocukları olan (0….n/2) ve (n/2+1….n-1) aralıklarını inceleyip hangisinin bizim aradığımız aralıkla kesiştiğini bulacağız. Kesişmeler için 2 durum mevcuttur birincisi çocuklardan sadece bir tanesi bu aralıkla kesişiyordur ve ikincisi her ikisi de kesişiyordur.

İlk durum kolay: kesişen çocuğa git ve aranan aralık yine sorgulanan aralığın aynısı olacak şekilde şu an açıklanan metodu oradan uygulamaya başla.

İkinci durumda ise öncelikle sol çocuğa git ve orada bu metodu uygula ardından sağ çocuğa git ve orada bu metodu uygula her biri için bulunan sonuçların toplamı bizim root aralığımızda bulunan ve istediğimiz aralıktaki sayıların toplamını bize verecektir. Bu sırada sol çocuk eğer (l1,r1) aralığı ile kesişiyorsa ve sağ çocuk da (l2,r2) aralığı ile kesişiyorsa l2=r1+1 olacaktır ve çocuklar için sorgulanan aralıklar bunlar ile güncellenecektir.

Bu recursive(özyinelemeli) fonksiyon her defasında çocuklarını çağırarak sonucu arayacaktır ancak her zaman yeniden çağırmasına gerek kalmayacaktır. Diğer bir değişle önceden hesapladığımız aralık toplamlarını kullanacağız. Daha açık bir ifadeyle eğer istenilen aralık şuan sorgulamaya çalıştığımız aralığın tamamıyla kesişiyorsa bu durumda çocuklarından elimize ulaşacak verilere bakmaksızın önceden hesapladığımız değeri kullanabiliriz.

Neden çalışma zamanı O(log n)? Sorgu işlemi sırasında her seviyede en fazla 4 node'u incelediğimizi düşünelim bu durumda ağacın yüksekliğinin log n olduğunu biliyoruz ve bu yüzden toplam çalışma zamanımız yaklaşık olarak yine log n olacaktır.

Şimdi her bir seviyede en fazla 4 node'u inceleyeceğimizi gösterelim. Başlangıç seviyesinde sadece root node ile ilgileniyoruz ikinci adımda ise en kötü durumda 2 node ile ilgileniriz ancak elimizde şöyle bir bilgi mevcut sağ çocukta bizle kesişen aralığın başlangıç indisi sol çocukta bizle kesişen aralıktaki bitiş indisinden bir fazla. Bu da bize bir sonraki seviye için ipucu vermektedir. Diğer seviyede en kötü durumda 2. seviyedeki node'lardan ikişer defa bu işlemi çağırırız. Bu durumda oluşan 4 çocuğun

durumlarını inceleyecek olursak bu çocuklardan bir tanesinden tekrar dallanmamıza gerek yoktur. Çünkü bu çocukların taşıdıkları aralıkların ya bir tanesi bizim sorguladığımız aralıkla kesişmiyordur yahut iki aralık kesişiyordur. Birinin kesişmediği durumda kesişmeyen aralıkla ilişkimiz sonlanmıştır yani daha fazla oradan dallanmayacağız. İkisinin de kesiştiği durumda aradığımız aralık bir tanesini bütün olarak içinde bulunduruyor demektir bu durumda önceden hesapladığımız değeri kullanırız ve tekrar dallanmayız. Son durumda çalışma zamanımız O(log n) olacaktır.

Güncelleme

Yapmamız gereken şey yine A dizisinin i. indisindeki sayıyı x değeri ile güncellemek yani A[i]=x.

Yapımızın hızını korumak açısından bu işlemi yine O(log n) zamanda yapmalıyız. Öncelikle şunu görmeliyiz ki A[i] sadece az sayıda aralığın içerisinde bulunuyor yani yaklaşık log n adet.

Ardından yapacağımız işlem bariz tıpkı oluşturma sırasında yaptığımız gibi yaprak node'dan başlayıp root node' dek bütün aralıkların hepsini tek tek güncellemeliyiz ve bunu yine recursive bir fonksiyonla rahat bir şekilde yapabiliriz.

Kodlama

Ana olarak yapmamız gereken şey verileri nasıl saklayacağımıza karar vermektir ve burada ufak bir incelik yapıp ağacı açık açık tree olarak değil de array üzerinde saklayacağız. Root node 1 ile temsil edilecek ve onun çocukları sırasıyla sol ve sağ çocuğu 2 ve 3 olacak. Ve genel halde ise i numaralı node'un çocuklar 2i ve 2i+1 olacaktır.

Bu metod veri yapısının kodlanmasını aşırı derecede kolaylaştıracaktır. Şimdi yapmamız gereken uzun uzadıya pointerlar vs kullanmadan arrayde bir kaç küçük toplama işleminden ibaret olacaktır.

Ancak önemli bir ayrıntı arrayin boyutu 2n olmayacaktır. Çünkü bu veri yapısı için array in ilk 2n indisinde hiç kullanmayacağımız elemanların da olması muhtemeldir. Bu yüzden n değeri için boyutu n den büyük ikinin ilk kuvvetine yuvarlamalıyız ve bunun için uzun işlemler yapmak yerine 4n boyutunda bir dizi bizim için yeterlidir.

Segment tree t[] adlı arrayin içerisinde saklanacaktır.

```
int n, t[4*MAXN];
```

Oluşturma metodu parametre olarak bir a dizisi alacaktır ardından güncellenecek olan alt ağacın root elemanını belirten bir v ve bu node'un aralığını belirten tl ve tr değişkenlerini alacaktır. Başlangıçta program parametre olarak v=1,tl=0,tr=n-1 alacaktır.

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

Sorgulama işlemi yine recursive bir fonksiyon ile tanımlanmıştır. Yukarıda verilen fonksiyonla aynı şekilde veriler root'tan çocuk node'lara aktarılacaktır. Program ilk çağrıldığında v,tl,tr değerleri sırasıyla 1,0,n-1 olacatır ve l ve r değerli bize sorgulanan aralığı göstermektedir.

```
int sum (int v, int tl, int tr, int l, int r) {
	if (l > r)
		return 0;
	if (l == tl && r == tr)
		return t[v];
	int tm = (tl + tr) / 2;
	return sum (v*2, tl, tm, l, min(r,tm))
		+ sum (v*2+1, tm+1, tr, max(l,tm+1), r);
}
```

Son olarak güncelleme işlemi. Bu işlem yine yukarıdakilere benzer şekilde recursive olarak tanımlanmıştır. Yeni olarak pos değişkeni dizi üzerinde değişecek olan indisi ve new_val değişkeni oraya atanacak olan yeni değeri temsil etmektedir.

```
void update (int v, int tl, int tr, int pos, int new_val) {
	if (tl == tr)
		t[v] = new_val;
	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] = t[v*2] + t[v*2+1];
```

```
        }
}
```

Ayrıca güncelleme işlemini recursive olmayacak şekilde de kodlamak mümkündür çünkü güncelleme işlemi aslında sadece bir kez başka bir işlemi çağırmaktadır. Ayrıca bu şekilde kodlayarak bu işlemi hızlandırmak da mümkündür.

Bunun yanında bir kaç küçük optimizasyon daha mevcuttur örneğin sayısal olarak bölme yerine bitwise operatörleri kullanılabilir ve bu da yine hızda bir miktar artmayı sağlayacaktır.

Karmaşık Segment Tree Versiyonları

Segment tree - aşırı esnek veri yapısıdır ve bir çok başka işlemin yapılmasına müsaittir. Şimdi onların bir kaçından bahsedelim.

Minimum ve Maksimum

Veri yapımızda küçük bir değişiklik yaparak aralıklar için toplamı bulduğumuz durumdan aralık sorguları için o aralıktaki maksimumu ya da minimumu bulmaya dönüşecektir. Bu işlemi yapmak için oluşturma ve güncelleme işlemleri sırasında fonksiyondan döndürülen değer toplam değil aralıkların maksimumu ve minimumu şeklinde değişecektir.

Minimum, maksimum ve onların bulunma sayıları

Problem bir öncekine çok benzer ancak maksimum veya minimum değerine ek olarak bulunma sayıları da hesaplanacaktır. Bu problem aslında bir başka problemin çözümü olarak ortaya çıkmıştır: verilen bir array üzerinden artan sırada olan en uzun alt dizilerin sayısını bulmak.

İki elemanın birleştirilmesi için ayrı bir fonksiyon gerçekleştireceğiz çünkü bu işlemi hem sorgulama hem de güncelleme sırasında kullanacağız.

```
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]);
	}
}
```

EKOK ve EBOB sogulanması

Yapmamız gereken şey aralıklar için sorguladığımız max ya da min işlemi yerine yine ebob ve ekok işlemleriyle yenileyeceğiz. Bu durumda yine sorgulama, oluşturma ve sorgulama işlemlerinde max ve min işlemlerin yerini ebob ve ekok ile yenileyeceğiz.

Sıfırların sayısını sayma ve K. sıfırı bulma

Bu tarz bir işlemi gerçekleştirmek için yine oluşturma,toplam ve güncelleme işlemlerini kullanalım. Eğer yapıda sıfır olan elemanlar için segment tree üzerindeki o pozisyonu 1 ile güncellersek toplam olarak elimize gelecek olan veri kaç adet sıfır olacağını buluruz.

K. sıfırı bulmak için yapmamız gereken sorgulama işleminde ufak bir değişikliktir. Bu işlemle root'tan başlayarak çocuklara dallanacağız ancak dallanma sırasında ilk önce sol çocukla ilgilenelim eğer sol çocukta bulunan sıfır sayısı K'den büyük veya eşit ise bu durumda aradığımız sıfır sol çocuk içerisindedir. Eğer küçükse bu durumda sadece sağ çocukla ilgilenmemiz yeterlidir çünkü sol çocuk içerisinde bulunma ihtimali yoktur.

Kodlanmasını inceleyelim: aşağıdaki kod parçası K. sıfırın yerini bulmak için kullanılabilir ve eğer K'den az sayıda 0 mevcutsa bu durumda -1 cevap olarak dönecektir.

```
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]);
}
```

İstenilen toplamdan büyük ilk prefixi bulma

Bizden istenen işlem verilen bir A dizisi için toplamı X ten büyük toplamdan büyük en kısa prefixin uzunluğunu bulmak olsun. Bu durumda yapmamız gereken şey bir önceki işlemimize çok benzemektedir.

Bu problemi öncelikle binary search ve sorgulama kullanarak yapabiliriz ancak bu durumda çalışma zamanımız $O(\log^2 n)$ olacaktır.

Binary search işlemi yapmak yerine bu işlemi tek bir sorgulama içersinde de yapmak mümkündür. Yapmamız gereken root node'dan dallanırken eğer sol çocuk toplam olarak istenilen toplamdan daha büyük ise sadece sol çocuk ile devam etmek aksi halde sadece sağ çocuk ile devam etmek mantıklıdır. Ve bu şekilde istenilen işlemi O(log n) çalışma zamanında yapabiliriz.

Maksimum toplama sahip aralığı bulma

Bizden istenen elimizde olan bir A dizisi üzerinde (0….n-1) aralığında ki maksimum toplama sahip olan (l,r) aralığını bulmaktır.

Aslında bu diğerlerinin aksine biraz daha karışık bir genellemedir. Segment tree üzerindeki her bir node için tek değer değil de dört adet değer saklayacağız. Bunlar sırasıyla: temsil edilen aralığın toplamı, maksimum toplama sahip prefix, maksimum toplama sahip suffix, temsil edilen aralığın maksimum toplama sahip alt aralığın değeri yani istenilen değer.

Ağacı nasıl oluşturacağımızı inceleyelim: yine recursive bir metot geliştireceğiz. Asıl inceleyeceğimiz şey sol ve sağ çocuğun nasıl birleştireceğini belirlememizdir. Bir node için altındaki sol ve sağ çocuğun durumlarına göre oluşabilecek bir kaç durum mevcuttur:

- Sol çocuk içinden seçilecek maksimum toplamlı aralık şu an bulmak istenilen aralığın en büyük toplamlı aralığı olabilir.
- Sağ çocuk içinden seçilecek maksimum toplamlı aralık şu an bulmak istenilen aralığın en büyük toplamlı aralığı olabilir.
- Diğer bir ihtimal ise sol çocuğun maksimum suffix değeri ile sağ çocuğun maksimum prefixinin toplamı yine bize bir alt dizi verecektir ve bu oluşak alt dizi yeni maksimum aralığa sahip alt diziyi oluşturabilir.

Sonuç olarak cevap aranan aralığın maksimum değeri bu üç durumun maksimumu olacaktır. Ayrıca maksimum suffix toplamı ve maksimum prefix toplamı kolayca hesaplanabilir. Şimdi sol ve sağ çocuktan gelen veriler ile bulunulan node'un yeni değerlerinin nasıl hesaplanacağını inceleyelim.

```
struct data {
	int sum, pref, suff, ans;
};

data combine (data l, data r) {
	data res;
	res.sum = l.sum + r.sum;
	res.pref = max (l.pref, l.sum + r.pref);
	res.suff = max (r.suff, r.sum + l.suff);
	res.ans = max (max (l.ans, r.ans), l.suff + r.pref);
	return res;
}
```

Güncelleme ve oluşturma işlemlerin nasıl yapılacağını inceleyelim. Bizim yapacağımız yaprak node'lara inildiği sırada onların tüm değerlerini rahatça hesaplayabiliriz ve bu değeri biz make_data metoduyla yapılacaktır.

```
data make_data (int val) {
	data res;
	res.sum = val;
	res.pref = res.suff = res.ans = max (0, val);
	return res;
}

void build (int a[], int v, int tl, int tr) {
	if (tl == tr)
		t[v] = make_data (a[tl]);
	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]);
	}
}

void update (int v, int tl, int tr, int pos, int new_val) {
	if (tl == tr)
		t[v] = make_data (new_val);
	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]);
	}
}
```

Geriye sadece sorguları cevaplama kaldı. Yine basit olarak yaptığımız sorgulama işlemi gerçekleştireceğiz ve sorgu olarak verilen aralıkla kesişen aralıkların birleştirmesiyle oluşacak data için elimize gelen maksimum toplama sahip aralık değeri bizim aradığımız cevap olacaktır.

```
data query (int v, int tl, int tr, int l, int r) {
      if (l == tl && tr == r)
            return t[v];
      int tm = (tl + tr) / 2;
      if (r <= tm)
            return query (v*2, tl, tm, l, r);
      if (l > tm)
            return query (v*2+1, tm+1, tr, l, r);
      return combine (
            query (v*2, tl, tm, l, tm),
            query (v*2+1, tm+1, tr, tm+1, r)
      );
}
```

Bir aralık için altındaki bütün verileri saklama

Bu metodun en kolay hali belli bir aralıktaki bütün değerleri o aralığı temsil eden node'un içinde saklamaktır ve bunu da en kolay vector, map, set veri yapılarıyla yapabiliriz. Yani çoğunlukla sıralı bir liste halinde tutmak işimizi kolaylaştıracaktır.

Bu durum için ilk akla gelen soru ne kadar hafıza gerekli olacak? Bu veri yapısı için gerekli olan hafıza O(n log n) dir. Çünkü her bir seviye için bu seviyedeki bütün node'lar üzerinde saklanacak veri sayısı toplamda O(n) dir ve ağacın yüksekliği O(log n) ve son durumda saklanacak veri sayısı O(n log n) olur.

Şimdi bu veri yapısını kullanabileceğimiz bir kaç problemi inceleyelim.

Belli bir aralıkta belli bir sayıdan büyük eşit en küçük sayıyı bulma

Sorgu olarak verilen (l,r,x) değerleri için l ve r indisleri arasında x'ten büyük veya x'e eşit en küçük sayının değerini bulmamız istenmektedir.

Bir segment tree oluşturacağız ve her elemanında bir sıralı liste barındıracak. Öncelikle yine recursion bir metot kullanacağız. Yine alt parçaları oluşturduktan sonra sol ve sağ çocuğu kullanarak bulunulan node'un temsil ettiği aralığı bulmaya çalışacağız. Bunu iyi bir şekilde yapmak için birleştirme işlemini linear zamanda yapmamız gerekmektedir kısaca açıklamak gerekirse elimizde var olan sıralı diziyi her ikisinin başında bulunan pointerları adım adım taşıyarak toplamda iki dizideki eleman sayısı kadar işlem yaparak birleştirilmiş halindeki sıralı diziyi oluşturabiliriz. .Bunun için de C++ içindeki STL'i kullanarak yapabiliriz. Buradaki vector veri yapısını ve hazır merge işlemini kullanabiliriz ve bu işlemi linear zamanda gerçekleştirmiş oluruz.

```
vector<int> t[4*MAXN];

void build (int a[], int v, int tl, int tr) {
	if (tl == tr)
		t[v] = vector<int> (1, a[tl]);
	else {
		int tm = (tl + tr) / 2;
		build (a, v*2, tl, tm);
		build (a, v*2+1, tm+1, tr);
		merge (t[v*2].begin(), t[v*2].end(), t[v*2+1].begin(),
t[v*2+1].end(),
			back_inserter (t[v]));
	}
}
```

Sorgulama sırasında yapacağımız işlem önceki sorgulamalara ek olarak sorgulayacağımız aralığın içinde yer alan aralıkların her birinde x'ten büyük ilk sayıyı binary search işlemi ile bulmak olacaktır. Bir adet binary search işlemi yeterlidir çünkü elimizdeki diziler sıralıdır ve rahatça bulabiliriz.

Bu işlemin toplamda çalışma zamanı O($\log^2$ n) olacaktır. Toplamda O(log n) adet işlemi (l,r) aralığındaki aralıkları belirlemek için yaparız ve her birinde de O(log n) işlemi binary search için gerçekleştiririz.

```
int query (int v, int tl, int tr, int l, int r, int x) {
	if (l > r)
		return INF;
	if (l == tl && tr == r) {
		vector<int>::iterator pos = lower_bound (t[v].begin(),
t[v].end(), x);
		if (pos != t[v].end())
			return *pos;
		return INF;
	}
	int tm = (tl + tr) / 2;
	return min (
		query (v*2, tl, tm, l, min(r,tm), x),
		query (v*2+1, tm+1, tr, max(l,tm+1), r, x)
	);
}
```

INF büyük bir sayıyı temsil etmektedir. Bunun sebebi sorgulama sırasında fonksiyonun bulunmayan bir aralık için çağrıldığında verilecek cevabın diğer cevaplara etkisini yok etmektir.

Belli bir aralıkta belli bir sayıdan büyük eşit en küçük sayıyı bulma(Güncelleme mümkün)

Soru bir öncekine benzer sadece bu soruda dizinin belli bir elemanını güncellemek mümkün yani A[i]=x işlemi mevcut.

Bu soru için az önceki işlemde kullandığımız vector yerine yine STL'de bulunan multiset veri yapısını kullanacağız. Oluşturma işlemi ve sorgulama işlemi yine öncekilere benzer olacaktır burada sadece güncelleme işlemini göstereceğiz. Güncelleme sırasında güncellenecek indisin birim aralığını tutan node'dan root node'a kadar olan bütün aralıktaki verilerin arasından eski değeri çıkarıp yeni değeri eklememiz gerekmektedir.

Burada güncelleme işlemi yine sorgulamaya benzer şekilde $O(\log^2 n)$ çalışma zamanına sahip olacaktır. Bunun sebebi toplamda O(log n) adet node'un set'inde değişiklik yapacağız ve her bir değişiklik multiset'in bize sunduğu ölçüde toplamda O(log n) zaman alacaktır. Bu durumda çalışma zamanı totalde $O(\log^2 n)$ olacaktır.

```
void update (int v, int tl, int tr, int pos, int new_val) {
      t[v].erase (t[v].find (a[pos]));
      t[v].insert (new_val);
      if (tl != tr) {
            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);
      }
      else
            a[pos] = new_val;
}
```

Burada gösterilen vector ve multiset kullanılan örneklerin üzerine burada kullandığımız veri yapısını değiştirebilir ve bunların yerine bir segment tree daha kullanabilirizi ya da treap benzeri daha kompleks veri yapıları kullanabiliriz.

Aralık üzerindeki güncellemeler

Şimdiye kadar güncellemelerin dizi üzerindeki sadece bir noktayla olan problemlerle ilgilendik ancak aynı şekilde dizi üzerinde belli bir aralıkta yapılacak değişiklere de olanak verecek şekilde segment tree versiyonları da mevcuttur. Yine öncekilere benzer olarak bu işlemi de O(log n) zamanda gerçekleştirebiliriz.

Belli bir aralığı artırma

Bize verilen problem şu şekildedir: Verilen bir dizi üzerinde belli bir aralık için artırılma işlemi verilebilir örneğin (l,r,add) l ve r arasındaki bütün indislere add değerini eklemeliyiz. Ve bir indisteki elemanın belli bir andaki değeri sorgulanabilir

Bu durumda yapmamız gereken şey önceki örneklerdeki aralık sorgulama işlemindeki gibi bize artırılma işlemi için belirtilen aralığın içerdiği aralıkları bulmaktır. Ve bulunan bu aralıklar için burası ve altında bulunan tüm aralıklar için eklenen değeri bu node'un altına kaydetmektedir. Bu durumda O(n) işlemde tek tek diziyi güncellemek yerine O(log n) işlemde artırılacak aralıkları güncellemiş olduk.

Sorgu sırasında verilecek bir i indisi için i indisini birim aralık olarak içeren node'dan itibaren root'a kadar olan yoldaki node'ların add değerlerinin toplamı ve birim node'daki ilk değerin toplamı bize son durumda i indisindeki sayının değerini verecektir. Ve yine bu işlem önceki örneklerde de olduğu gibi O(log n) zaman alacaktır.

Kod örneği ile açıklama daha iyi anlaşılacaktır. Güncelleme sırasında her bir node için add değeri t arrayinde saklanacaktır. Ve oluşturma sırasında herbir yaprak node'un ilk değeri verilmiştir.

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

void update (int v, int tl, int tr, int l, int r, int add) {
      if (l > r)
            return;
      if (l == tl && tr == r)
            t[v] += add;
      else {
```

```
            int tm = (tl + tr) / 2;
            update (v*2, tl, tm, l, min(r,tm), add);
            update (v*2+1, tm+1, tr, max(l,tm+1), r, add);
      }
}

int get (int v, int tl, int tr, int pos) {
      if (tl == tr)
            return t[v];
      int tm = (tl + tr) / 2;
      if (pos <= tm)
            return t[v] + get (v*2, tl, tm, pos);
      else
            return t[v] + get (v*2+1, tm+1, tr, pos);
}
```

Belirli bir aralıktaki tüm sayıları değiştirme

Bu problem için iki çeşit işlem mevcuttur. Bunlar sırasıyla belli bir aralığı belli bir sayıya eşitleme ve belli bir indisteki sayıyı sorgulama olacaktır.

Çözümde her bir node için bu node'un temsil ettiği aralık belirli bir sayıya boyandı mı boyanmadı boyandıysa hangi sayı ile boyandığını tutmamız gerekmektedir. Bu bize aralıktaki bütün sayıları tek tek güncellemek yerine birkaç tane aralığı boyama imkanı sağlayacaktır. Boyalı olan node altındaki bütün aralığın belirli bir sayıya eşit olduğunu bize gösterecektir.

Örneğin, (0,n-1) aralığı belli bir sayıya eşitlenmiş olsun, bu durumda bizim güncelleme işlemimiz sadece root node'un boyanmış olduğu sayıyı değiştirecektir ve başka bir işlem yapmayacaktır. İkinci olarak dizinin ilk yarısının başka bir sayıya eşitlendiği durum gelmiş olsun. Bu durumda bizim dizinin ilk yarısını yeni gelen sayıyla boyamamız gerekmektedir ancak ondan önce bizim şuan root node'da duran sayı ile dizinin ikinci yarısını boyamamız gerekmektedir. Çünkü root node boyalı olarak kalamaz, eğer kalırsa bizim daha sonra gelecek olan sorgularımızı etkilemiş olur bu yüzden root node'un sağ ve sol çocuğunu boyamalı ve root node'un boyalı halini tekrar yok etmeliyiz. Bunu gerçekleştirdikten sonra sol çocuğu yeni gelen sayı ile boyayabiliriz.

İttirme işlemi: Recursive olarak güncelleme işlemini gerçekleştirdiğimiz sırada bulunduğumuz node'un üzerinde bir değişiklik yapılacağı zaman öncelikle bu node üzerinde boyalı bir sayı varsa onu değerlendirmeliyiz. Yani eğer bulunulan node bir boyanmış sayıya sahipse bu node'un boyanmış olduğu sayı değerini kaldırıp sol ve sağ çocuğunu o değer ile boyamalıyız. Ardından bize verilen güncelleme işlemini gerçekleştirmeliyiz.

Özet olarak: Her güncelleme için belirli birkaç aralığı boyuyoruz bu işlemler için bize gerekli olan çalışma zamanı O(log n) dir. Ayrıca her boyama için ağaç üzerinde gezdiğimiz sürede gezilen bütün notlar için ittirme işlemi yapıyor. Bu durum bizim çalışma zamanımıza etki etmiyor.

Şimdi bir kod ile örneğini inceleyelim:

```
void push (int v) {
	if (t[v] != -1) {
		t[v*2] = t[v*2+1] = t[v];
		t[v] = -1;
	}
}

void update (int v, int tl, int tr, int l, int r, int color) {
	if (l > r)
		return;
	if (l == tl && tr == r)
		t[v] = color;
	else {
		push (v);
		int tm = (tl + tr) / 2;
		update (v*2, tl, tm, l, min(r,tm), color);
		update (v*2+1, tm+1, tr, max(l,tm+1), r, color);
	}
}

int get (int v, int tl, int tr, int pos) {
	if (tl == tr)
		return t[v];
	push (v);
	int tm = (tl + tr) / 2;
	if (pos <= tm)
		return get (v*2, tl, tm, pos);
	else
		return get (v*2+1, tm+1, tr, pos);
}
```

get fonksiyonu başka şekilde de kodlanabilir. İndisin olduğu birim aralığı bulana kadar ittirme işlemini yapmaya gerek yoktur. Bunun yerine boyalı olarak bulunan ilk sayı o sayının son durumdaki değerini oluşturacaktır.

Belli bir aralığı artırma ve maksimum eleman sorgulama

Bizden istenen verilen bir A dizisi için verilen aralık artırma işlemlerini gerçekleştirmek ve ardından yapılacak olan belli bir aralıktaki maksimum elemanı bulma sorgularına cevap vermemizdir.

Problemin çözümü için ekstradan bir değer tutmamız gerekmektedir bu değer bu aralığın ne kadar artırıldığını temsil edecektir.

Varsayalım root'un sol çocuğunun bulunduğu aralık x kadar artırılmış olsun bu durumda bizim yapmamız gereken root node'unun altındaki maksimum elemanı tekrar hesaplamak olacaktır bunu da şu şekilde hesaplayabiliriz: sol çocuğun altındaki maksimum + sol çocuğun atlına eklenen ekstra değer ile sağ çocuğun altındaki maksimum + sağ çocuğun altına eklenen ekstra değer bize root node'unun kendi altındaki maksimumu verecektir. Bu şekilde her aralık artırıldığında aralığın içerdiği node'lardan bir kaçı güncellenecektir(içerdiği aralık tamamen onun içinde olan en büyük aralıklar yani tıpkı önceki sorgulardaki gibi) ve güncellenen değerlerle segment tree üzerindeki değerler yeniden oluşturulacaktır. Bu şekilde toplamda O(log n) zamanda belli bir aralığı artırabiliriz.

Güncelleme işlemiyle aynı şekilde istenilen aralık için segment tree'deki içerdiği aralıkların maksimumunu bulabiliriz ve bu maksimumlara root'tan o node'a ulaşana dek gezdiği node'lardaki ekstraları ekleyerek eklemeler dahil maksimum değerini yine O(log n) çalışma zamanında bulabiliriz.

2 boyutlu Segment Tree

Problem şudur: verilen bir 2 boyutlu A[0….n-1][0….m-1] matrisi için 2 tip işlemi gerçekleştirmeliyiz bunlardan bir tanesi belirli bir dikdörtgen alandaki değerlerin toplamının sorgusu ve diğeri de belli bir noktaya bir değer atanması şeklinde olacaktır.

Oluşturma işlemi: daha anlaşılır olması için iki boyutlu halini kullanmaktansa tek boyutlu gibi düşünmemiz daha iyi olacaktır. Normal tek boyutta bir segment tree oluşturacağız bu segment tree x koordinatları için oluşmuş olacak. Ancak normal segment tree gibi her bir node için bir değer saklamayacağız her bir node'un altında bir segment tree daha oluşturacağız ve bu oluşan segment tree y koordinatları için oluşacak. Aşağıdaki kod anlaşılmasına yardım edecektir.

```
void build_y (int vx, int lx, int rx, int vy, int ly, int ry) {
      if (ly == ry)
            if (lx == rx)
                  t[vx][vy] = a[lx][ly];
            else
                  t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
      else {
            int my = (ly + ry) / 2;
```

```
            build_y (vx, lx, rx, vy*2, ly, my);
            build_y (vx, lx, rx, vy*2+1, my+1, ry);
            t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
      }
}

void build_x (int vx, int lx, int rx) {
      if (lx != rx) {
            int mx = (lx + rx) / 2;
            build_x (vx*2, lx, mx);
            build_x (vx*2+1, mx+1, rx);
      }
      build_y (vx, lx, rx, 1, 0, m-1);
}
```

Oluşan segment tree yine linear hafıza kullanacaktır toplam kullanılan hafıza 16nm olacaktır.

Sorgu işlemi sırasında oluşturma işlemindeki gibi yine yapıyı önce tek boyutlu olarak hayal edebiliriz. Tek boyutta istenilen x koordinatındaki aralıklarının içindeki aralıklara bakacağız tıpkı üstte yaptığımız gibi. Ardından bakılması gereken aralıkların her birinde y koordinatındaki aralıkları aranacak ve orada saklanan değerler ile istenilen veriye ulaşılacaktır.

```
int sum_y (int vx, int vy, int tly, int try_, int ly, int ry) {
      if (ly > ry)
            return 0;
      if (ly == tly && try_ == ry)
            return t[vx][vy];
      int tmy = (tly + try_) / 2;
      return sum_y (vx, vy*2, tly, tmy, ly, min(ry,tmy))
            + sum_y (vx, vy*2+1, tmy+1, try_, max(ly,tmy+1), ry);
}

int sum_x (int vx, int tlx, int trx, int lx, int rx, int ly, int
ry) {
      if (lx > rx)
            return 0;
      if (lx == tlx && trx == rx)
            return sum_y (vx, 1, 0, m-1, ly, ry);
      int tmx = (tlx + trx) / 2;
      return sum_x (vx*2, tlx, tmx, lx, min(rx,tmx), ly, ry)
```

```
            + sum_x (vx*2+1, tmx+1, trx, max(lx,tmx+1), rx, ly,
ry);
}
```

Bu fonksiyon O(log n * log m) işlem gerçekleştirecektir. Bunu standart segment tree üzerindeki yaptığımız sorgulama işlemlerini iki defa iç içe yaptığımızı düşünebiliriz.

Güncelleme işlemi: verilen bir x,y noktasına P değerini yazmak istiyorsak yani A[x][y]=P işlemini gerçekleştireceksek yapmamız gereken önce x koordinatını barındıran segment tree'nin birinci boyutundaki tüm aralıkların her birinin içindeki y boyutunu barından aralıklar için P değerini güncellememiz gerekmektedir. Bu da daha önceden yaptığımız tek boyutta nokta güncelleyip aralık toplamı sorgusunun aynısıdır.

```
void update_y (int vx, int lx, int rx, int vy, int ly, int ry,
int x, int y, int new_val) {
      if (ly == ry) {
            if (lx == rx)
                  t[vx][vy] = new_val;
            else
                  t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
      }
      else {
            int my = (ly + ry) / 2;
            if (y <= my)
                  update_y (vx, lx, rx, vy*2, ly, my, x, y,
new_val);
            else
                  update_y (vx, lx, rx, vy*2+1, my+1, ry, x, y,
new_val);
            t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
      }
}

void update_x (int vx, int lx, int rx, int x, int y, int
new_val) {
      if (lx != rx) {
            int mx = (lx + rx) / 2;
            if (x <= mx)
                  update_x (vx*2, lx, mx, x, y, new_val);
            else
```

```
                update_x (vx*2+1, mx+1, rx, x, y, new_val);
        }
        update_y (vx, lx, rx, 1, 0, m-1, x, y, new_val);
}
```

Geçmiş güncellemelerin saklandığı Segment Tree(Persistent Segment Tree)

Bu veri yapısı ile Segment Tree üzerindeki yapılan bütün değişiklikleri saklayarak ilerliyoruz. Bu bizim geçmişte olan bir durum için yapılabilecek sorguların cevaplarını vermemize olan sağlıyor. Tabi ki bu işlemi gerçekleştirirken bütün veri yapısını kopyalamayacağız.

Segment Tree yapısında yapılan güncellemeler sadece O(log n) değişiklik içermektedir. Ayrıca bunlar root node'dan değişikliğin yapıldığı yaprak node'a kadar olan yol boyunca olan node'lardır. Sadece bu değişen node'ları saklarız ve kalanı ise eski halinin aynısıdır.

En basit halinin kodlanışı aşağıdadır.

```
struct vertex {
        vertex * l, * r;
        int sum;

        vertex (int val)
                : l(NULL), r(NULL), sum(val)
        { }

        vertex (vertex * l, vertex * r)
                : l(l), r(r), sum(0)
        {
                if (l)  sum += l->sum;
                if (r)  sum += r->sum;
        }
};

vertex * build (int a[], int tl, int tr) {
        if (tl == tr)
                return new vertex (a[tl]);
        int tm = (tl + tr) / 2;
        return new vertex (
                build (a, tl, tm),
                build (a, tm+1, tr)
```

```
        );
}

int get_sum (vertex * t, int tl, int tr, int l, int r) {
        if (l > r)
                return 0;
        if (l == tl && tr == r)
                return t->sum;
        int tm = (tl + tr) / 2;
        return get_sum (t->l, tl, tm, l, min(r,tm))
                + get_sum (t->r, tm+1, tr, max(l,tm+1), r);
}

vertex * update (vertex * t, int tl, int tr, int pos, int
new_val) {
        if (tl == tr)
                return new vertex (new_val);
        int tm = (tl + tr) / 2;
        if (pos <= tm)
                return new vertex (
                        update (t->l, tl, tm, pos, new_val),
                        t->r
                    );
        else
                return new vertex (
                        t->l,
                        update (t->r, tm+1, tr, pos, new_val)
                    );
}
```