Lowest Common Ancestor(En Yakın Ortak Ata)

Bir ağaçta iki düğümün en yakın ortak ata'sını bulma problemi ile daha çok 20. yüzyılın 2. yarısında uğraşıldı ve şimdi temel graph algoritmalarının içinde bulunuyor. LCA bir çok string işleme ve bilgisayarsal biyoloji alanında kullanılan bir algoritmadır. Örneğin suffixtree ve ağacımsı veri yapıları sırasında kullanılmaktadır. Harel ve Tarjan lineer bazı işlemler yaptıktan sonra sabit zamanda sorguların cevaplanabileceğini göstermiştir. O zamandan beri bu yöntem genişletildi ve bu rehberde bu yöntemin yanında bazı kullanımlarını da göstereceğiz.

Daha az soyut bir örnekle başlayalım. Dünyadaki bir çok türün başka türlerin evrimiyle ortaya çıktığı bilinmektedir. Bu sistemi bir ağaçla gösterebiliriz. Her düğüm bir türü ve o düğümün çocukları da o türden evrimleşmiş türleri gösterir. Bu ağaçta iki düğümün LCA'sını bularak en yakın ortak atalarını buluruz ve benzer özelliklerinin bu atadan geldiklerini görürüz.

Range Minimum Query(RMQ) bir dizideki iki indis arasındaki minimum elemanı bulmaya yarayan bir algoritmadır. İlerleyen kısımlarda LCA'nın RMQ'ya indirgenebileceğini göstereceğiz.

Ancak RMQ, LCA'dan başka yerlerde de kullanılmaktadır. Örneğin Suffix Array(String aramayı Suffix Tree algoritması kadar hızlı yapan ve daha az yer kullanan bir algoritma) sırasında.

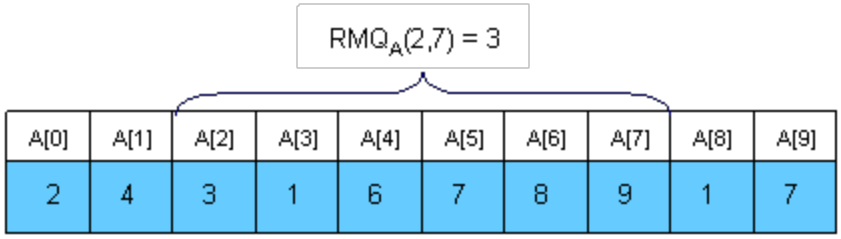
Bu rehberde önce RMQ hakkında konuşacağız. Bu problemi çözmek için birçok yöntem inceleyeceğiz. Daha sonra LCA-RMQ ilişkisini açıklayacağız.

Gösterimler

Önişleme maliyeti f(n) ve sorgu maliyeti g(n) olan bir yöntemi **< f(n), g(n) >** olarak göstereceğiz.
Bir A dizisinin i ve j indisleri arasındaki en küçük elemanı **RMQ*A*(i, j)** şeklinde göstereceğiz.
Bir T ağacındaki u ve v düğümlerinin en yakın ortak atasını da **LCA*T*(u,v)** şeklinde göstereceğiz.

Range Minimum Query(RMQ)

Bir A[0,N-1] dizisi ve iki indis verildiğinde iki indis arasındaki minimum elemanı bulma.



RMQ için verimsiz algoritmalar

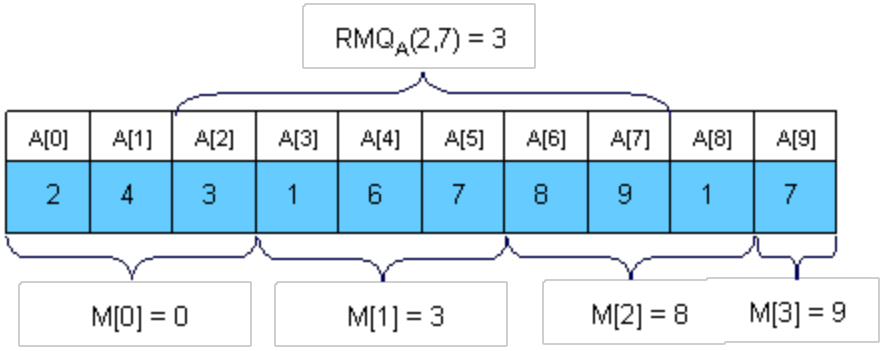
Her (i,j) ikilisi için **RMQ*A*(i,j)** değerini bir M[0,N-1][0,N-1] dizisinde kaydedebiliriz. Bu yöntemle maliyetimiz **<O(N^3), O(1)>** olur. Basit bir dinamik programlama kullanarak bunu **<O(N^2), O(1)>** 'ye indirebiliriz. M dizisini yaklaşık olarak şöyle hesaplarız:

```
void işlem1(int M[MAXN][MAXN], int A[MAXN], int N)
{
    int i, j;
      for (i =0; i < N; i++)
      M[i][i] = i;
      for (i = 0; i < N; i++)
      for (j = i + 1; j < N; j++)
                if (A[M[i][j - 1]] < A[j])
                  M[i][j] = M[i][j - 1];
                else
                M[i][j] = j;
}
```

Bu yöntem **O(N^2)** hafıza kullanır ve yavaş çalışır bu yüzden büyük girdiler için çalışmaz.

<O(N), O(sqrt(N))> çözüm

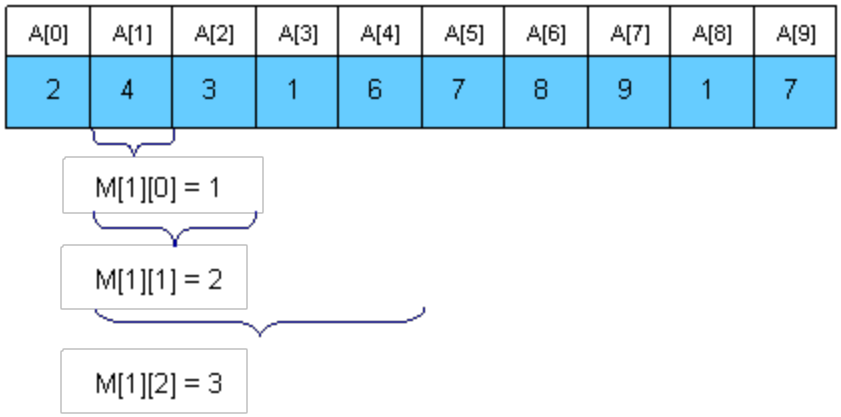
Bu yöntemde ise diziyi kök(N) lik parçalara ayıracağız. Ve her kök(N) lik parçanın minimumunun yerini tutan M[0,kök(N)-1] dizisine kaydedeceğiz. M dizisini **O(N)** maliyetle doldurabiliriz. M dizisi şekildeki gibi olacaktır:



Şimdi bunu kullanarak **RMQ*A*(i, j)** yi nasıl hesaplayacağımıza bakalım. Tamamı (i,j) aralığında olan kök(N) lik parçaları direk alırız. Geriye ise (i,j) aralığının başında ve sonunda bazı elemanlar kalır. Bunları da iteratif olarak hesaplarız. Örneğin yukarıdaki dizi için **RMQ*A*(2, 7)** 'yi hesaplıyor olalım. Cevabımız şu olur Minimum(A[2], A[M[1]], A[6], A[7]). Bu algoritma her sorgu için maksimum 3*kök(N) işlem yapar. Bu yöntemin temel avantajları hızlı kodlanması(Topcoder, Codeforces gibi yarışmalar için) ve problemin dinamik hali(Sorgular arasında dizinin değiştirilmesi) için de kullanılabilmesi.

Sparse Table (Ayrık Tablo) çözümü

Bu problem için bir diğer yöntem ise dinamik programlama kullanarak 2^k uzunluğunda alt dizileri hesaplamak ve kullanmak. Bir M[0,N-1][0, logN] dizisini dolduracağız ve M[i][j] i'den başlayan 2^j uzunluğundaki altdizinin en küçük elemanının yerini tutacak. Şu şekilde:



M[i][j] 'yi hesaplarken (i,i +2^j -1) aralığının ilk yarısına ve ikinci yarısına bakacağız. Başka bir deyişle i'den ve i+2^{j-1} den başlayan 2^{j-1} lik parçalara bakacağız. Formülize edersek :

$$M[i][j] = \begin{cases} M[i][j-1], & A[M[i][j-1]] <= A[M[i+2^{j-1}-1][j-1]] \\ M[i+2^{j-1}-1][j-1], & otherwise \end{cases}$$

M dizisini yaklaşık olarak şöyle hesaplarız:

```
void işlem2(int M[MAXN][LOGMAXN], int A[MAXN], int N)
 {
    int i, j;

  //1 uzunluğundaki altdizileri doldurma
    for (i = 0; i < N; i++)
     M[i][0] = i;
  //küçük aralıkları kullanarak büyükleri hesaplama
     for (j = 1; 1 << j <= N; j++)
     for (i = 0; i + (1 << j) - 1 < N; i++)
       if (A[M[i][j - 1]] < A[M[i + (1 << (j - 1))][j - 1]])
          M[i][j] = M[i][j - 1];
            else
              M[i][j] = M[i + (1 << (j - 1))][j - 1];
  }
```

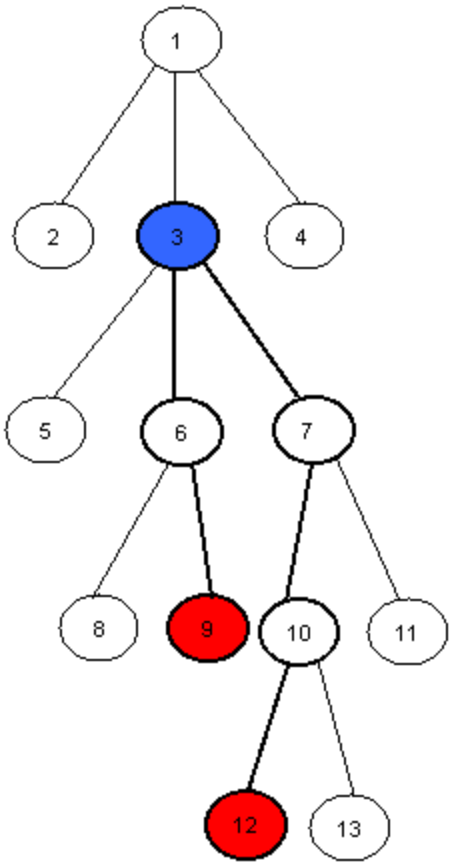
Bu değerler elimizde olduğunda **RMQ*A*(i, j)** 'yi hesaplarken ise birleştiğinde (i,j) aralığını kaplayacak 2 tane aralığın minimumu cevabımız olur. Örneğin **RMQ*A*(2, 10)** için Minimum(M[2][3], M[10-2^3+1][3]). Formulize hali ise şöyledir:

$$RMQ_A(i,j) = \begin{cases} M[i][k], & A[M[i][k]] <= A[M[j-2^k+1][k]] \\ M[j-2^k+1][k], & otherwise \end{cases}$$

Bu yöntemle ise maliyetimiz **<O(N logN), O(1)>** olur.

Lowest Common Ancestor

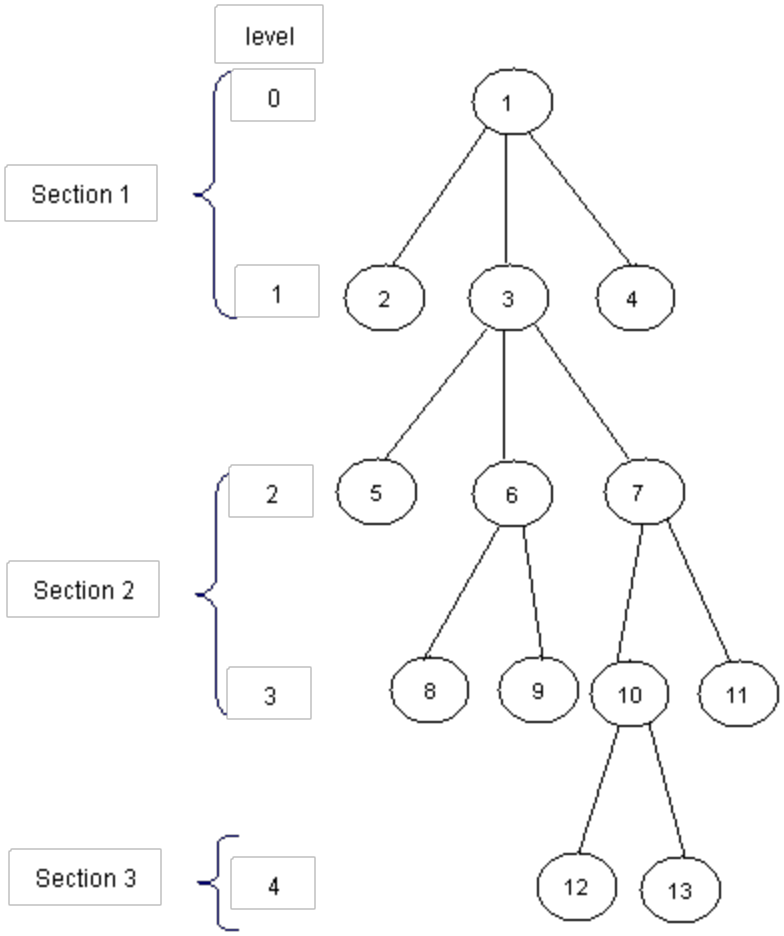
Bir T ağaçı ve (u,v) düğüm ikilisi verildiğinde kök düğüme en uzak iki düğümün de ortak atası olan düğüm bu iki düğümün LCA'sıdır. Örnek:



$LCA_T(9, 12) = 3$

<O(N), O(kök(N))> çözüm

Girdiyi kök(N)'lik parçalara bölmek RMQ problemini çözerken izlediğimiz bir yöntemdi. Bu yöntemi LCA problemine de uyarlayabiliriz. Bu sefer ağacı kök(H)'lik parçalara böleceğiz(H ağacın yüksekliği). 1. parça yüksekliği (0, kök(H)-1) seviyelerini, 2. parça (kök(H),2*kök(H)-1) seviyelerini içerecek vb. . Parçalara bölünmüş bir ağaç örneği:



Artık her düğümün bir önceki parçanın en alt seviyesindeki atasını biliyoruz. Bu değerleri bir P[1,MAXN] dizisine kaydedeceğiz. Yukarıdaki ağaç için P dizisi şöyle olmalı(En üst parçadaki düğümler için önceki grubun atasını kök düğüm kabul edelim):

P[1]	P[2]	P[3]	P[4]	P[5]	P[6]	P[7]	P[8]	P[9]	P[10]	P[11]	P[12]	P[13]
1	1	1	1	3	3	3	3	3	3	3	10	10

Her parçanın en üst seviyesinde olan düğümler için P[i] o düğümün ilk atası olacaktır. Yani P[i] = T[i] diyebiliriz(T[i] = i düğümünün 1. atası.). P dizisini dfs ile şöyle doldurabiliriz (L[i] = i düğümünün seviyesi):

```
void dfs(int node, int T[MAXN], int N, int P[MAXN], int L[MAXN], int nr)  {
    int k;

 //Eğer düğüm 1. parçadaysa
 //P[düğüm] = 1
 //Eğer düğüm bir parçanın
 //en üst seviyesindeyse P[düğüm] = T[düğüm]
 //İki durum da geçerli değilse
 //P[düğüm] = P[T[düğüm]]
    if (L[düğüm] < kök(H))
        P[düğüm] = 1;
    else
       if(!(L[düğüm] % kök(H)))
           P[düğüm] = T[düğüm];
       else
           P[düğüm] = P[T[düğüm]];

   for k sırasıyla bulunduğumuz düğümün bütün çocuklarıolmak şartıyla
       dfs(k, T, N, P, L, nr);
 }
```

Bu aşamadan sonra kolaylıkla sorgu yapabililriz. LCA(x,y) 'yi bulmak için önce LCA'larının hangi parçada olduğunu bulduktan sonra o parçada seviye seviye yukarı çıkarak LCA'larını buluruz. İşte kod:

```
int LCA(int T[MAXN], int P[MAXN], int L[MAXN], int x, int y)
 {
 //öncelikle x ve y'yi aynı parçaya getiriyoruz
 //bunun için daha derinde olanı P dizisindeki değerine eşitliyoruz(bir önceki parçaya taşımış
oluyoruz)
    while (P[x] != P[y])
       if (L[x] > L[y])
          x = P[x];
       else
```

```
        y = P[y];
  //Artık aynı parçada olduklarına göre seviye seviye tırmandırarak makismum kök(H) işlemde
//LCA'yı buluruz
    while (x != y)
      if (L[x] > L[y])
      x = T[x];
      else
      y = T[y];
      return x;
  }
```

Bu fonksiyon en fazla 2*kök(H) işlem yapar ve maliyetimiz **<O(N), O(sqrt(H))>** olur. H(yükseklik) en kötü durumda N'e eşit olacağından maliyetimiz **<O(N), O(sqrt(N))>** olur. Bu algoritmanın en büyük avantajı hızlı kodlanabililr olmasıdır.

<O(N logN, O(logN)> lik çözüm

Bu problemi daha hızlı çözmek istediğimizde ise Dinamik Programlama kullanabiliriz. Bir P[1,N][1,logN] dizisi (P[i][j] i düğümünün 2^j . atasını tutuyor). Bu diziyi doldurmak için aşağıdaki özyinelemli fonksiyonu kullanabiliriz:

$$P[i][j] = \begin{cases} T[i], & j = 0 \\ P[P[i][j-1]][j-1], & j > 0 \end{cases}$$

Yazacağımız fonksiyon ise takriben şöyle olur:

```
void işlem3(int N, int T[MAXN], int P[MAXN][LOGMAXN])
 {
      int i, j;

  //P dizisini -1'lerle doldururuz
      for (i = 0; i < N; i++)
      for (j = 0; 1 << j < N; j++)
            P[i][j] = -1;

  //her i düğümünün 20. atası T[i]'ye eşit
```

```
    for (i = 0; i < N; i++)
    P[i][0] = T[i];

//bottom up dynamic programing
   for (j = 1; 1 << j < N; j++)
      for (i = 0; i < N; i++)
        if (P[i][j - 1] != -1)
           P[i][j] = P[P[i][j - 1]][j - 1];
}
```

Bu işlemin zaman ve bellek maliyeti **O(N logN)** ‘dir. Şimdi sorguları nasıl yapacağımıza bakalım. L[i] = i düğümünün ağaçtaki seviyesi olsun. Eğer p ve q düğümleri aynı seviyedeyse meta-binary search(Meta-binary search 2’nin kuvvetlerinin kullanıldığı binary search) kullanarak LCA(p,q)’yu hesaplayabiliriz. 2’nin her j. kuvveti (j log(L[p])’den 0’a azalarak gidiyor) Eğer P[p][j] != P[q][j] ise LCA’nın daha yukarı seviyelerde olduğunu biliriz ve p = P[p][j] ve q = P[q][j] diyip LCA(p,q)’yu hesaplarız. Böyle devam ederse p ve q aynı düğümün çocuğu olduğunda duracağız ve T[p] cevabımız olacak. Aynı seviyede değiller ise daha derinde olanı benzer bir yöntemle diğerinin seviyesine çıkarabiliriz. Query fonksiyonu takriben böyle olmalı:

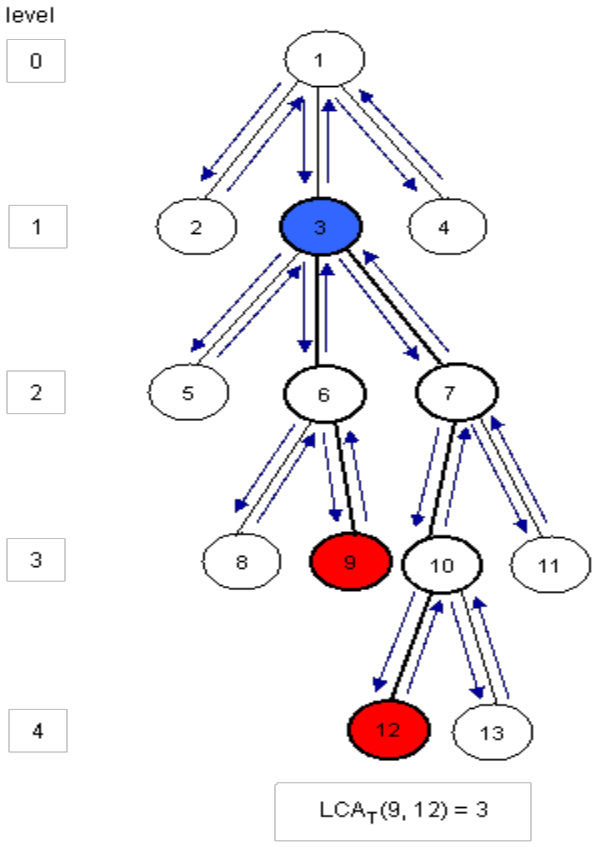
```
int query(int N, int P[MAXN][LOGMAXN], int T[MAXN],
 int L[MAXN], int p, int q)
 {
      int tmp, log, i;

  //Eğer q p’den daha derindeyse değiştirelim
    if (L[p] < L[q])
       tmp = p, p = q, q = tmp;

 //[log(L[p)] değerini hesaplayalım
    for (log = 1; 1 << log <= L[p]; log++);
      log--;

 //p’yi q ile aynı seviyedeki atasına eşitleyelim
      for (i = log; i >= 0; i--)
        if (L[p] - (1 << i) >= L[q])
        p = P[p][i];
      if (p == q)
      return p;
```

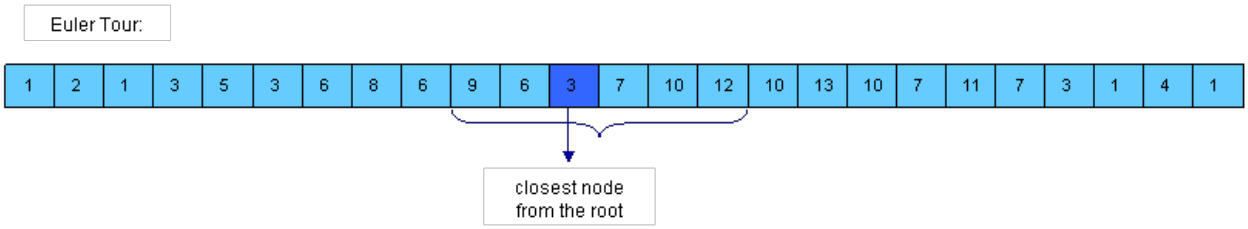
```
 //P deki değerleri kullanarak LCA(p, q)'yu hesaplayalım
    for (i = log; i >= 0; i--)
      if (P[p][i] != -1 && P[p][i] != P[q][i])
        p = P[p][i], q = P[q][i];
      return T[p];
 }
```

Bu fonksiyon en fazla 2 * logH işlem yapar. En kötü durumda H = N olacağından maliyetimiz **<O(N logN), O(logN)>** olur. Bu yöntemi hem kodlaması kolay hem de öncekinden daha hızlı çalışıyor.

LCA yı RMQ ya indirgeme

LCA sorguları için RMQ'yu nasıl kullanacağımıza bakalım. Aslında yaptığımız LCA'yı lineer zamanda RMQ'ya dönüştürmek. Bunu yaptıktan sonra her RMQ algoritmasını LCA için de kullanabiliyor olacağız. Nasıl yapacağımızı bir örnekle açıklayalım:



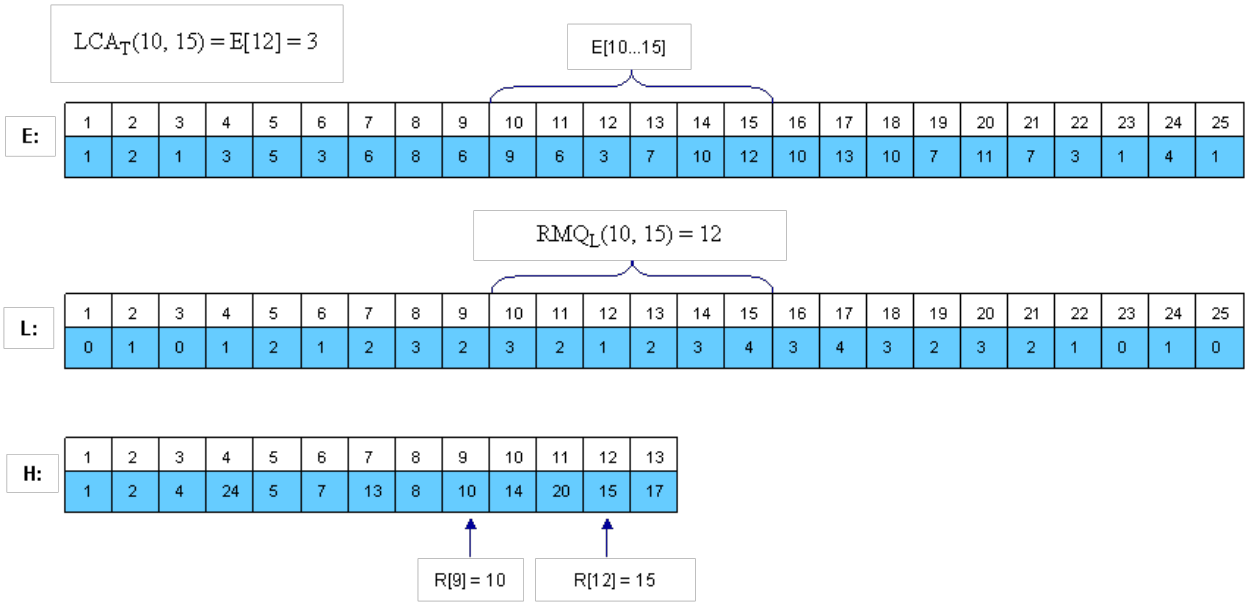


Euler tour: Ağacı dfs'yle gezdiğimizde uğradığımız her düğümü yazarak elde ettiğimiz dizi. herhangi bir u'dan v'ye giderken LCA'larından yukarıda hiçbir düğüme uğramayacak olmasını kullanacağız.

LCA*T*(u, v) ağacı dfs ile gezerken u ve v'ye uğramalarımız arasındaki kök düğüme en yakın olan düğüme eşittir. Dolayısıyla euler tour'la elde ettiğimiz dizi de u ve v arasındaki en düşük seviyedeki düğümü bularak **LCA*T*(u, v)** 'yı bulabililriz. Bunun için 3 adet diziye ihtiyacımız var:

- E[1,2*N-1] - Ağacı dfs ile gezerken elde ettiğimiz dizi. E[i] i. sırada uğradığımız düğüm
- L[1,2*N-1] - Ağacı dfs ile gezerken uğradığımız her düğümün seviyeleri. E[i]. seviyesi
- H[1,N] - H[i] E dizisinde i'nin ilk göründüğü yer

H[u]'nun H[v]'den küçük olduğunu varsayalım(değilse u ve v'yi değiştiririz). u düğümüyle v düğümü arasında uğradığımız düğümler E[H[u]] , E[H[u]+1] … E[H[v]] dir. Şimdi yapmamız gereken bu aralıkta seviyesi en düşük olan düğümü bulmak. Bunun için de RMQ kullanacağız. Yani **LCA*T*(u, v) = E[RMQ*L*(H[u], H[v])]**. E,H ve L dizisi aşağıdaki gibi olmalıdır.



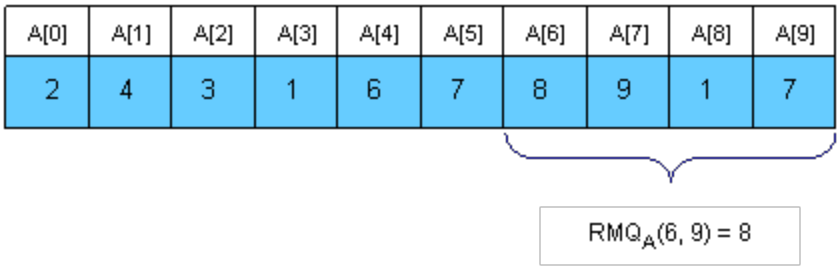
L dizisinde ardışık 2 eleman arasındaki farkın daima 1 olduğuna dikkat ediniz.

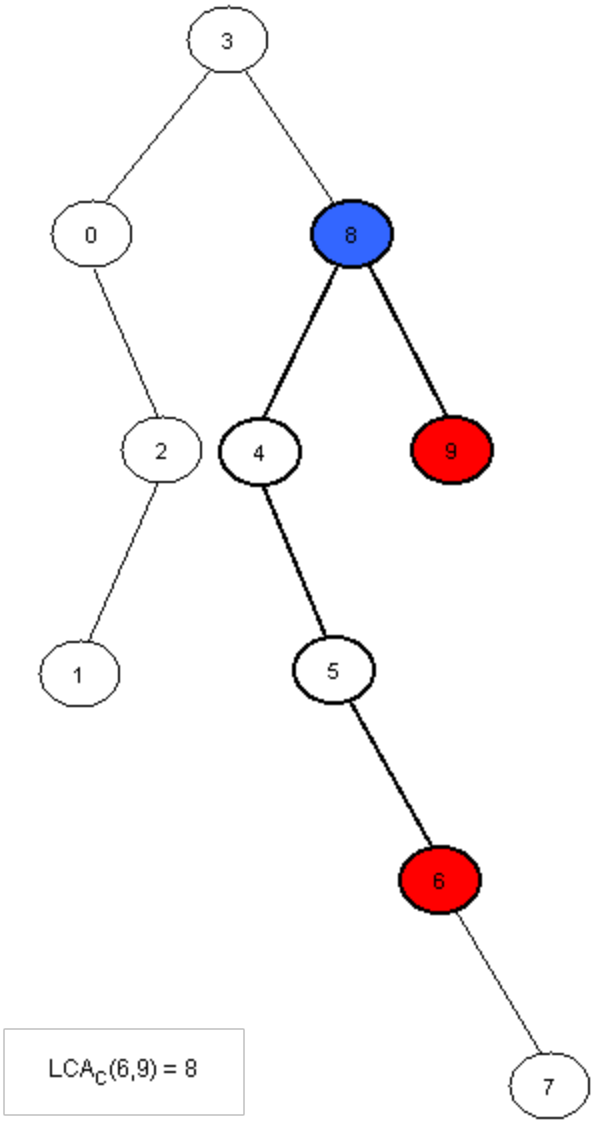
RMQ'dan LCA'ya

LCA probleminin RMQ'ya lineer zamanda indirgenebileceğini gördük. Şimdi de RMQ problemini LCA'ya nasıl indirgeyebileceğimizi göreceğiz. Bu demektir ki genel RMQ problemini kısıtlı hale dönüştürebiliriz(dizide ardışık iki eleman arasındaki fark 1 ise). Bunun için de Cartesian Tree'leri kullanacağız.

Cartesian Tree bir A[0,N-1] dizisinin ağaç halidir. C(A) ağacının kötü A dizisinin minimum elemanıdır. i minimum elemanın bulunduğu indis olsun; i > 0 ise kök düğümün sol alt ağacı A[0,i-1]'in Cartesian Tree halidir. i <= 0 ise sol çocuğu yoktur. Sağ çocuk da aynı şekilde A[i+1,N-1]'in Cartesian Tree halidir. Eğer A dizisi aynı değerde elemanlar içeriyorsa birden fazla Cartesian Tree olabilir. Bu rehberde minimum değerin ilk göründüğü yer alınacağından bir değerden birden fazla olmayacaktır. Bu durumda **RMQ*A*(i, j) = LCA*C*(i, j)** olacaktır.

Örnek:





Geriye bir tek C(A)'yı lineer zamanda hesaplamak kalıyor. Bunu bir stack kullanarak yapabililriz. Başta stack boş olacak. i. adımda stack'e A[i] elemanını ekleyeceğiz ama bundan önce A[i]'den büyük bütün elemanları çıkaracağız. A[i]'yi eklediğimiz yerde çıkarma işlemlerinden önce bulunan eleman i'nin sol çocuğu olacak, A[i] de ondan önceki elemanın sağ çocuğu olacak. Her

adımda stack'ın en başındaki eleman C(A)'nın kökü olacak. Stack'te elemanların değeri yerine indislerini tuttuğumuzda ağaç yapısını kurmak daha kolay olacak.

Stack'ın nasıl çalışacağına dair bir örnek:

Adım	Stack	Ağaçta yapılan işlemler
0	0	0 ağaçtaki tek düğüm.
1	0 1	1 stack'e eklendi. Artık, 1 0'ın sağ çocuğu.
2	0 2	2 0'ın yanına eklendi, ve 1 silindi (A[2] < A[1]). Artık 2 0'ın sağ çocuğu ve 2'nin sol çocuğu 1.
3	3	A[3] şimdiye kadarki en küçük eleman, o yüzden bütün elemanlar stack'ten silindi ve 3 kök oldu. 3'ün sol çocuğu 0.
4	3 4	4 3'ün yanına eklendi, 3'ün sağ çocuğu 4.
5	3 4 5	5 4'ün yanına eklendi, 4'ün sağ çocuğu 5.
6	3 4 5 6	6 5'in yanına eklendi, 5'in sağ çocuğu 6.
7	3 4 5 6 7	7 6'nın yanına elendi, 6'nın sağ çocuğu 7.
8	3 8	8 3'ün yanına eklendi, daha büyük elemanlar silindi. 8 3'ün

		sağ çocuğu oldu ve 8'in sol çocuğu 4.
9	3 8 9	9 8'in yanına eklendi, 8'in sağ çocuğu 9.

A daki her elemanın sadece bir defa eklenilebilip bir defa çıkarılabileceğine dikkat ediniz. Bu işlemin maliyeti **O(N)**. Ağacı oluşturma fonksiyonunu şu şekilde oluşturabiliriz:

```
void computeTree(int A[MAXN], int N, int T[MAXN])
{
	int st[MAXN], i, k, top = -1;

 //boş bir stack'le başlıyoruz
 //i. adımda stack'e A[i] elemanını ekleyeceğiz
	for (i = 0; i < N; i++)
	{
 //A[i]'den küçük eşit
 //ilk elemanın yerini buluyoruz
	k = top;
	while (k >= 0 && A[st[k]] > A[i])
		k--;
 //Ağacı yukarda anlatıldığı gibi kuruyoruz
	 if (k != -1)
		T[i] = st[k];
	 if (k < top)
		T[st[k + 1]] = i;
 //A[i]'yi stack'e ekliyoruz
 //ve daha büyük elemanları siliyoruz
	st[++k] = i;
	top = k;
	}
 //Stack'teki ilk eleman kök düğüm olacak
 //O yüzden herhangi bir düğümün altında değil
	T[st[0]] = -1;
 }
```

Bazı RMQ'lar için <O(N), O(1)> maliyetli bir algoritma

Artık özel bazı RMQ problemlerinin LCA'ya dönüştürülebileceğini biliyoruz. RMQ yapacağımız dizideki ardışık iki eleman arasındaki fark daima 1 ise **<O(N), O(1)>** ile çalışan algoritmayı kullanabiliriz. Artık problemimiz bir A[0, N -1] dizisinde RMQ bulmak öyle ki A dizisinin ardışık 2 elemanı arasındaki fark daima 1. A dizisini N-1 elemanlı şöyle bir binary diziye dönütüreceğiz.

A[i] = A[i] - A[i + 1]. Artol A dizisindeki elemanlar sadece -1 ya da +1 olabilir. Böylece A[0]'dan A[i]ye kadar olan elemanların toplamı A[i]'nin eski değerini veriyor olacak. Yine de bundan sonra eski değerlere ihtiyacımız olmayacak.

Bundan sonra problemi çözmek için A dizisini L = logN / 2 'lik parçalara böleceğiz. ~A[i] = A dizisinin i. parçasındaki en küçük eleman olsun. B[i] de bu minimum elemanın A dizisindeki indisini tutsun. ~A ve B dizisi N/L elemandan oluşuyor. ~A dizisini yukarıda anlatılan Sparse Table yöntemini kullanarak oluşturabiliriz. Bunun maliyeti **O(N/L * log(N/L)) = O(N)** olur. Bu işlemden sonra parçaları kapsayan kısımlar için sorguyu **O(1)** de yapabiliriz. Geriye parçaları kapsamayan kısımlarda arama yapmak kalıyor. Her parçanın boyutu logN(oldukça küçük) ve bunun aynı zamanda bir binary dizi olduğuna da dikkat edelim. L uzunluğunda toplam 2^L = kök(N) adet binary dizi olabilir. Yani L uzunluğundaki her parçadaki her indis ikilisi için P dizisine bakarız. Bu işlemin zaman ve bellek maliyeti **O(kök(N)*L^2)=O(N**) olur. P'yi oluştururke A dizisindeki her parçanın türünü bir T[1, N/L] dizisinde tutabiliriz. Parçanın türü -1'i 0'la, +1'i de 1'le göstererek elde edilir.

Bu adımlardan sonra **RMQ*A*(i, j)** için iki durumumuz var:

- i ve j aynı parçadadır, bu durumda hesaplamayı P ve T dizilerini kullanarak yaparız.
- i ve j farklı parçalardadır. Bu durumda da 3 değer hesaplarız: i'den i'nin bulunduğu parçanın sonuna kadar olan minimum, i ve j'nin bulunduğu parçaların minimumları(bunu ~A dizisiyle yapabiliriz) ve j'nin bulunduğu parçanın başından j'ye kadar olan minimumu tekrar P ve T dizilerini kullanarak hesaplayabiliriz.

Sonuç

RMQ ve LCA birbirlerine benzeyen, dönüştürülebilen problemlerdir. İkisini de çözmek için bir çok yöntem vardır.

Bazı LCA ve RMQ problemleri için:

http://acm.pku.edu.cn/JudgeOnline/problem?id=1986

http://acm.pku.edu.cn/JudgeOnline/problem?id=2374

http://acmicpc-live-archive.uva.es/nuevoportal/data/problem.php?p=2045

http://acm.pku.edu.cn/JudgeOnline/problem?id=2763

http://www.spoj.pl/problems/QTREE2/

http://acm.uva.es/p/v109/10938.html

http://acm.sgu.ru/problem.php?contest=0&problem=155

Kaynakça

- "Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE" [PDF] by Johannes Fischer and Volker Heunn

- "The LCA Problem Revisited" [PPT] by Michael A.Bender and Martin Farach-Colton - a very good presentation, ideal for quick learning of some LCA and RMQ aproaches

- "Faster algorithms for finding lowest common ancestors in directed acyclic graphs" [PDF] by Artur Czumaj, Miroslav Kowaluk and Andrzej Lingas