Ağaçta Set-Swap Yöntemi

C++'daki hazır swap() fonksiyonunun zaman karmaşıklığı O(1)'dır. İki arrayi hatta iki set'i bile swapleseniz fark etmez. Bu özelliği bazı ağaç sorularında zaman karmaşıklığını azaltmakiçin kullanacağız. Örnek bir soru üzerinden gidelim:

N tane şehir ve bu şehirleri bağlayan N-1 tane yol var, yani graphın yapısı bir ağaç şeklinde. 1 numaralı node başkenttir. Yollar tek yönlüdür. Her şehre tam olarak X uzaklığında kaç tane şehir olduğunu bulmak istiyoruz(yollar her zaman başkentte uzaklaşacak şekildedir). İlk başta şöyle bir çözüm düşünün: bir map arrayimiz olacak, bu arrayde her node için o nodeun altındaki nodeların başkente olan uzaklıkları tutulacak(bu nodeun altında başkente a uzaklığında b tane node var şeklinde). Problemi özyinelemeli(recursive) olarak çözdüğümüzü düşünelim, o an bulunduğumuz node için alt ağacındaki bütün mesafeler biliniyor olacak biz de bize lazım olan mesafeyle çözümümüzü güncelleyip bütün çözümleri yukarı aktaracağız. Bu şekilde düşününce zaman karmaşıklığı $O(N^2)$ gibi görünmüş olabilir ama küçük bir değişiklikle $O(N\log^2 N)$'e indireceğiz. Önce çözüm fonksiyonunu gözden geçirelim:

dist[] dizisinde her nodeun başkente olan uzaklığını tutuluyor
g[] vector dizisinde her nodedan çıkan edgeler var
mp[] dizisinde her nodeun altındaki her uzaklıktaki yerlerin sayısı tutuluyor(pair gibi (uzaklık,sayı) şeklinde)

```
typedef pair<int,int> ii;

inline void dfs( int nd ){

	for( vector<ii>::iterator it=g[nd].begin() ; it!=g[nd].end() ; it++ ){

		dfs(it->fi);

		if( sz(mp[it->fi])>sz(mp[nd]) )		// Onemli bolum burasi
			swap(mp[it->fi],mp[nd]);

		for( map<int,int>::iterator it2=mp[it->fi].begin() ; it2!=mp[it->fi].end() ; it2++ )
			mp[nd][it2->fi]+=it2->se;

		res[nd]=mp[nd][dist[nd]+D];
	}

	mp[nd][dist[nd]]++;
}
```

mp[nd] kısmında nd numaralı nodeun alt ağacının bazı kollarının hesaplanmış bir şekilde bulunduğunu düşünün. it->fi ise sıradaki kolun başlangıç nodeu ve dfs(it->fi)'yi çağırdığımızda it-fi'den sonraki kısmın çözümü mp[it-fi]'ye kaydedilmiş oldu. mp[it-fi]'de en fazla it-fi'nın alt ağacındaki node sayısı kadar sayıda veri olabilir(tam olarak değil, en fazla; çünkü aynı uzaklıkta birden fazla node olduğunda bunlar aynı yerde tutulabiliyor). mp[it-fi]'den gelen verileri mp[nd]'ye işlememiz lazım ki yukarı düzgün bir şekilde aktarabilelim. İşte burada swap foksiyonu devreye giriyor. Eğer mp[it-fi]'de daha çok veri varsa mp[nd] ile mp[it-fi]'nin yerlerini değiştiriyoruz ki az sayıda olan veriyi çok sayıda olanın üzerine yazabilelim. Bunun zaman karmaşıklığını iki uç durum üzerinden düşünebiliriz:

İlk durum ağacın zincir hâlinde olduğu durumdur, yani her nodeun en fazla bir tane çocuğu var. Bu durumda her zaman aşağıdan gelen çözüme eklenecek tek node o an bulunduğumuz node olacaktır. Aşağıdan kaç tane çözüm gelmiş olursa olsun aşağıdaki kümeyle o an bulunduğumuz nodeun kümeleri yer değişir(çünkü başlangıçta bizim kümemiz boş) sonra da aşağıdaki küme(yeni hâli) gezilir(içinde hiç eleman olmayacak). Fonksiyonun sonunda da asıl kümemize o anki node eklenir ve sonuç yukarıya dönülür. Zincir durumunda Her node için sadece bir kere ekleme yapacağız.

İkinci durum ise ağacın tamamen dengeli olması durumudur. Her nodeun K tane çocuğu olduğunu varsayalım. Her nodeun ilk kolunun çözümü doğrudan swaplenecek ve diğer kollar teker teker aktarılacak(duruma göre bunlara da swap uygulanacak). Yani herhangi bir node en fazla ağacın derinliği kez aktarılacak. Derinliğin en fazla olduğu durum ise K'nin 2 olduğu durumdur. Her seviyede alttan gelen nodeların yarısı swap sayesinde doğrudan yukarıya geçer, çünkü her nodeun iki çocuğu var ve bu çocukların ilkinden gelen sonuçlar doğrudan swaplenecek. Zaman karmaşıklığını üstteki fonksiyona göre düşündüğümüzde ilk for döngüsü toplamda N defa içerideki for döngüsü ise ortalama logN sayıda dönecek, içteki forun içindeki aktarma işlemi de yaklaşık logN. Zaman karmaşıklığı $O(Nlog^2N)$ oldu.

Bazı problemde istenene göre çözümleri yukarıya aktarırken lower_bound(), find() vb. fonksiyonları kullanmamız gerekebilir. Bununla ilgili bir soruyu inceleyelim.

IOI 2011 - Race

N tane şehir ve bunları birbirine bağlayan N-1 tane yol var(yine ağaç yapısı). Yolların uzunlukları da var ve yollar çift yönlü. Bizden istenen ise tam olarak K uzunluğunda bir yarış pisti inşa etmemiz. Bu pist aynı nodeu veya edgei birden fazla kez kullanamaz. Bu pisti en az kaç tane edge kullanarak inşa edebileceğimiz soruluyor. Doğrudan kod üzerinden gidelim:

```
#define fi first
#define se second
```

```
#define sz(a) ((int)a.size())

typedef pair<Lint,int> li;

inline void dfs( int nd , int pre , Lint cur ){

	for( vector<ii>::iterator it=g[nd].begin() ; it!=g[nd].end() ; it++ )
		if( it->fi!=pre ){

			dep[it->fi]=dep[nd]+1;
			dfs(it->fi,nd,cur+it->se);

			set<li>::iterator it2=st[it->fi].lower_bound(li(cur+k,-1));

			if( it2!=st[it->fi].end() && it2->fi==cur+k )
				res=min(res,it2->se-dep[nd]);

			if( sz(st[it->fi])>sz(st[nd]) )
				swap(st[nd],st[it->fi]);

			for( it2=st[it->fi].begin() ; it2!=st[it->fi].end() ; it2++ ){
				set<li>::iterator it3=st[nd].lower_bound(li(k-it2->fi+2*cur,-1));
				if( it3!=st[nd].end() && it3->fi==k-it2->fi+2*cur )
					res=min(res,it3->se+it2->se-2*dep[nd]);
			}

			for( it2=st[it->fi].begin() ; it2!=st[it->fi].end() ; it2++ )
				st[nd].insert(*it2);

		}
		st[nd].insert(li(cur,dep[nd]));
}
```

Bu problemde iki durumu kontrol ediyoruz:

- O anda bulunduğumuz nodeun alt ağacında kendisine K uzaklıkta olan nodelar
- x+y=K için o anda bulunduğumuz nodeun bir kolunda kendisine x uzaklığında bulunan bir node ve başka bir kolunda kendisine y uzaklığında bulunan başka bir node.

İlk durum için:

```
set<li>::iterator it2=st[it->fi].lower_bound(li(cur+k,-1));

if( it2!=st[it->fi].end() && it2->fi==cur+k )
        res=min(res,it2->se-dep[nd]);
```

lower_bound'u li(cur,-1) üzerinden çalıştırıypruz çünkü en az edge içeren çözümü arıyoruz.

İkinci durum için:

```
for( it2=st[it->fi].begin() ; it2!=st[it->fi].end() ; it2++ ){
        set<li>::iterator it3=st[nd].lower_bound(li(k-it2->fi+2*cur,-1));
        if( it3!=st[nd].end() && it3->fi==k-it2->fi+2*cur )
                res=min(res,it3->se+it2->se-2*dep[nd]);
}
```

O an çözümünü aldığımız koldan(it->fi numaralı nodeun bulunduğu kol) gelen her node için daha önceki kollardaki nodelardan herhangi birine uzaklığı K mi diye kontrol ediyoruz.

Bu iki duruma göre res(cevabımız res değişkeninde tutuluyor, bu değişkeni ilk başta INT_MAX'a eşitleyebilirsiniz)i güncelledikten sonra yeni koldan gelen nodelari şimdiki nodeumuzun setine ekliyoruz. En son o an bulunduğumuz nodeu da sete ekleyip fonksiyonu bitiriyoruz.