BÖLÜM 7

ALGORİTMANIN ETKİNLİK ANALİZİ

ANAHTAR KAVRAMLAR

Etkinlik Analizi, Zaman Karmaşıklığı, Fonksiyon Büyüme Hızı, En – İyi Durum, En – Kötü Durum, Ortalama Durum, Asimptotik Notasyon

ÖĞRENME HEDEFLERİ

1. Algoritmanın etkinlik analizinin nasıl uygulanacağını bilir
2. Algoritmanın temel işleminin nasıl belirleneceğini bilir
3. Fonksiyonların büyüme hızının nasıl belirleneceğini bilir
4. Asimptotik notasyonları bilir
5. Asimptotik notasyonların algoritmaların etkinlik analizinde nasıl uygulanacağını bilir

YEDİNCİ BÖLÜM

GİRİŞ

Algoritmalar için iki tür etkinlik analizi göz önünde bulundurulmaktadır: zaman etkinliğinin analizi (ya da zaman karmaşıklığı) algoritmanın verilen girdi için ne kadar sürede ya da ne kadar adımda istenen çıktıyı ürettiğinin belirlenmesi, alan etkinliğinin analizi (ya da alan karmaşıklığı) algoritmanın verilen girdi için istenen sonucu üretirken bellekten ne kadarlık kısma ihtiyaç duyduğu şeklinde tanımlanabilir. Etkinlik analizi algoritmanın pratikte uygulanıp uygulanamayacağına dair fikir verirken, aynı zamanda aynı problem için geliştirilmiş farklı algoritmaları da kıyaslayabilmemize imkan sağlamaktadır. Bu bölümde verilen bir algoritmanın etkinlik analizinin nasıl yapıldığı örnekler üzerinden detaylıca incelenecektir.

1. ÖLÇÜ BİRİMİNİN BELİRLENMESİ

Verilen bir algoritmanın zaman karmaşıklığını hesaplarken, yani algoritmanın verilen girdi için ne kadar sürede ya da ne kadar adımda istenen çıktıyı ürettiğini ölçerken; ilk yapılması gereken bu ölçümün hangi birim kullanılarak yapılacağının belirlenmesidir. Bu noktada akla ilk gelen zaman karmaşıklığını saniye ya da dakika gibi zaman birimleri cinsinden ölçmek olacaktır. Bunun için algoritmayı üzerinde çalıştırabileceğimiz bir bilgisayar bulmamız ve algoritmayı bu bilgisayarda çalıştırabileceğimiz bir programlama dilline aktarmamız gerekmektedir. Sonrasında verilen bir girdi için algoritmayı çalıştırabilir ve bir kronometre yardımıyla algoritmanın kaç saniyede istenen çıktıyı ürettiğini belirleyebiliriz. Burada elbette daha iyi donanıma sahip bir bilgisayar ya da programla dili için daha etkin bir derleyici kullanmak algoritmanın çalışma süresini kısaltacaktır. Dolayısıyla; bu ölçme yönteminde bilgisayarın sahip olduğu donanımın kalitesi, algoritmanın hangi programlama diline nasıl aktarıldığı ya da programlama dili için hangi derleyicinin kullanıldığı gibi etkenler algoritmanın çalışma süresini yani zaman karmaşıklığını doğrudan etkilemektedirler.

Diğer yandan, geliştirdiğimiz algoritmaları günlük pratikte kullanabilmemiz için uygun programlama dillerine aktarmamız ve belli bir takım donanım cihazları satın almamız, yani zaman ve para harcamamız gerekmektedir. Dolayısıyla zamanımızı ve paramızı boşa harcamamak için, daha henüz kağıt üstündeyken geliştirdiğimiz algoritmaların etkinlik analizinin yapılması ve belki de yatırıma değer olup olmadıklarının belirlenmesi gerekmektedir. Ve yukarıda ifade edilenler düşünüldüğünde bu analizin, algoritmaların üzerinde çalıştırılacağı bilgisayarın sahip olduğu donanımın kalitesi, algoritmanın hangi programlama diline nasıl aktarıldığı gibi etkenlerden bağımsız bir şekilde yapılması daha doğru bir yöntem olacaktır.

Algoritmanın zaman karmaşıklığını ölçerken uygulayabileceğimiz bir yöntem de, algoritmanın istenen çıktıyı üretirken gerçekleştirdiği bütün işlemleri belirlemek ve bu işlemlerin algoritma sonlandırılıncaya kadar kaç defa gerçekleştirildiğini hesaplamak olabilir. Yalnız bu yöntemde algoritmanın gerçekleştirdiği bütün işlemleri belirlemek algoritmanın zaman karmaşıklığı hesabı için hem zor hem de gereksiz bir uğraş olacaktır. Onun yerine algoritmanın gerçekleştirdiği ve algoritmanın çalışma süresini doğrudan etkileyen en önemli ve temel işlemi belirlemek ve algoritma sonlanıncaya kadar bu işlemin kaç defa uygulandığını hesaplamak daha doğru bir yaklaşım olacaktır.

Şimdi bu yöntem üzerinden sözde kodu Şekil 7.1'deki gibi olan ve verilen bir tamsayı dizisinin verilen bir tamsayıyı içerip içermediğini bulmaya yarayan algoritmanın zaman karmaşıklığını hesaplamaya çalışalım. **Arama** algoritmasında **for** döngüsü içerisinde aranan tamsayının, dizinin sıradaki elemanına eşit olup olmadığını kontrol etmemize imkan veren 't_k = x' kıyaslaması algoritmanın temel işlemidir. Örneğin [4, 1, 6, 3] tamsayı dizisi ve 10 tamsayısı verildiğinde, algoritma 't_k = x' kod parçası yardımıyla sırasıyla dizinin elemanlarını 10 elemanı ile kıyaslayacak ve dizinin elemanları aranan tamsayıya eşit olmadığından '0' değerine dönecektir. Dikkat edilirse algoritma bu örneklem için 4 kıyaslama yapacaktır. Diğer yandan girdiyi [7, 21, 6, 5, 12, 8, 9, 17] tamsayı dizisi ve 11 tamsayısı olarak belirlersek; bu sefer algoritma girdi üzerinde 8 kıyaslama yapacak ve verilen tamsayı dizisi verilen 11 tamsayısını içermediğinden yukarıdaki örnekleme benzer şekilde '0' değerine dönecektir.

DİKKAT EDELİM

Algoritmanın etkinlik analizi algoritmaların üzerinde çalıştırılacağı bilgisayarın sahip olduğu donanımın kalitesi, algoritmanın hangi programlama diline nasıl aktarıldığı gibi etkenlerden bağımsız bir şekilde yapılmalıdır.

```
1   Arama([t1, ...,tn]; x)
2
3   girdi : n boyutlu bir tamsayı dizi ve bir x tamsayısı
4   çıktı : dizi x'i içeriyorsa, dizinin x'e eşit ilk elemanının
5   indeksi; değilse, 0
6
7   for (k = 1 to n)
8       if tk = x
9           return k
10  return 0
```

Şekil 7.1 Arama Algoritmasının Sözde Kodu

Burada da görülebileceği gibi algoritmanın performansı ya da çalışma süresi ile algoritmanın girdi boyutu arasında doğrudan bir ilişki vardır. Daha büyük boyutlu dizilerin verilen bir elemanı içerip içermediğini belirlemek küçük boyutlu girdilere nazaran daha çok süre alacaktır. Dolayısıyla algoritmanın gerçekleştirdiği en temel işlemin algoritma tarafından kaç defa gerçekleştirildiğini hesap eden fonksiyon, algoritmanın girdi boyutu üzerinde tanımlanacaktır. İlgili fonksiyon girdi boyutu üzerinde tanımlandığından, etkinlik analizinde öncelikli olarak girdi boyutunun belirlenmesi gerekmektedir. Örneğin yukarıdaki örnekte olduğu gibi verilen bir tamsayı dizisinin verilen bir tamsayıyı içerip içermediğini belirlerken girdi boyutu ilgili tamsayı dizisinin eleman sayısı iken, verilen iki matrisin çarpımında girdi boyutu ilgili matrislerin satır ve sütün sayıları olmaktadır. Verilen fonksiyonu T(n) notasyonu ile gösterirsek; n elemanlı bir tamsayı dizisinin verilen bir x tamsayısını içerip içermediğini belirleyen ve Şekil 7.1'de sözde kod ile gösterimi verilen algoritmanın çalışma süresini veren fonksiyon T(n) = n'dir.

2. EN-İYI DURUM, ORTALAMA DURUM VE EN-KÖTÜ DURUM ETKİNLİK ANALİZLERİ

Şekil 7.1'de verilen **Arama** algoritması yukarıda da gösterildiği gibi, [7, 21, 6, 5, 12, 8, 9, 17] tamsayı dizisi ve 11 tamsayısı için 8 kıyaslamada sonuçlanmakta ve '0' değerine dönmektedir. Halbuki algoritma aynı boyuttaki [12, 20, 13, 4, 11, 14, 9, 3] dizisi ve 12 tamsayısı için tek kıyaslamada sonuçlanacak ve '1' değerine dönecektir. Görüleceği üzere algoritmalar, aynı boyutta olsalar bile farklı yapıdaki girdiler için farklı zaman karmaşıklığına sahip olabilmekte ve farklı sayıda adımda sonuç üretebilmektedirler. Algoritmanın etkinlik analizi için bu farklı zaman karmaşıklıklarından hangisi dikkate alınacaktır? Gelin önce **Arama** algoritması üzerinden, verilen bir algoritma için bahsi geçen farklı etkinlik analizlerini belirleyelim.

[12, 20, 13, 4, 11, 14, 9, 3] dizisi ve 12 tamsayısı örnekleminde de gözlemlendiği üzere; eğer dizinin ilk elemanı aranan tamsayıya eşitse, girdi boyutu ne olursa olsun **Arama** algoritması tek kıyaslamada sonuçlanmakta ve '1' değerine dönmektedir. Bu durum elde edilebilecek en iyi zaman karmaşıklığı olduğundan en-iyi durum etkinlik analizi olarak tanımlanmaktadır. Bu şekilde bir n boyutlu girdi için algoritmanın zaman karmaşıklığı T(n) = 1 olacaktır.

Diğer yandan [7, 21, 6, 5, 12, 8, 9, 17] dizisi ve 11 tamsayısı örnekleminde de gözlemlendiği üzere; eğer dizi aranan tamsayıyı içermiyorsa, **Arama** algoritması dizinin sonuna kadar dizinin elemanlarını sırayla aranan tamsayı ile kıyaslamakta ve n kıyaslama sonucunda '0' değerine dönmektedir. Bu durum bu algoritma için elde edilebilecek en kötü zaman karmaşıklığı olduğundan en-kötü durum etkinlik analizi olarak tanımlanmaktadır. Dolayısıyla bu şekilde bir n boyutlu girdi için algoritmanın zaman karmaşıklığı T(n) = n olacaktır.

Bir diğer incelenmesi gereken etkinlik analizi, algoritmanın rastgele bir girdi üzerinde kaç

adımda sonuç ürettiğini belirleyen ve algoritmanın etkinlik analizi ile ilgili genel durumu ifade eden ortalama durum etkinlik analizidir. Yalnız ortalama durum etkinlik analizinin uygulanabilmesi için girdilerin farklı durumlarının olasılık dağılımlarının bilinmesi gerekmektedir. Ortalama durum analizinin nasıl uygulanacağını yine **Arama** algoritması üzerinden anlamaya çalışalım. Önce farklı durumlar için algoritmanın zaman karmaşıklıklarını belirleyelim. Aranan tamsayı, dizinin ilk elemanına eşit olduğunda zaman karmaşıklığı T(n) = 1 olduğunu daha önce belirlemiştik. Benzer şekilde aranan tamsayı dizinin ikinci elemanına eşitse algoritma iki kıyaslamada sonuçlanacağından, algoritmanın zaman karmaşıklığı T(n) = 2 olacaktır. Genelleyecek olursak; i = 1...n için, aranan tamsayı dizinin i. elemanına eşitse T(n) = i olacaktır. Ayrıca dizi aranan sayıyı içermediğinde, algoritmanın zaman karmaşıklığının T(n) = n olduğunu da daha önce belirlemiştik.

Algoritmanın yeterince girdi üstünde denendiğini varsayalım ve i = 1...n olmak üzere, bu girdilerden yüzde kaçında aranan sayının dizinin i. elemanına eşit olduğu veya yüzde kaçının aranan sayıyı içermediği belirlenmiş olsun. Bu durumda ortalama durum etkinlik analizini uygularsak; örneğin girdilerin tamamı aranan sayıyı içermiyorsa, algoritmanın zaman karmaşıklığı T(n) = n olacaktır. Eğer girdilerin tamamı aranan sayıyı içeriyorsa ve i = 1...n için aranan sayının dizinin i. elemanına eşit olma durumlarının sayıları birbirine eşitse; algoritmanın zaman karmaşıklığı farklı durumların zaman karmaşıklıklarının ortalaması olacağından, T(n) = (1 + 2 + ... + n) / n, yani T(n) = (n + 1) / 2 olacaktır.

Bu analizden anlaşılacağı üzere; her ne kadar ortalama durum etkinlik analizi algoritmanın performansı ile ilgili en doğru tespiti yapıyor olsa da, bu tür bir etkinlik analizi girdilerin farklı durumlarının olasılık dağılımlarının bilinmesini gerektirmektedir. Böyle bir dağılımı elde etmek veya farklı girdi durumları için verilen bir olasılık dağılımını doğrulamak da zordur. Dolayısıyla ortalama durum etkinlik analizini uygulamak en-kötü ve en-iyi durum etkinlik analizine nazaran daha çok çaba gerektirmektedir.

Diğer yandan; en-iyi durum etkinlik analizi, bu durumu oluşturacak girdilerin gelme olasılığı düşük olduğundan algoritmanın performansını değerlendirme noktasında çok da fikir vermeyecektir. Bununla birlikte, algoritmanın en uzun sürede sonuç ürettiği olası en kötü girdiler üzerinden etkinlik analizinin yapılması, performans açısından en kötü durumda bile algoritmanın nasıl çalıştığını gözlemlemeye imkan sağladığından, algoritmanın zaman karmaşıklığı için genellikle en-kötü durum analizi dikkate alınmaktadır.

DİKKAT EDELİM

Algoritmanın en uzun sürede sonuç ürettiği olası en kötü girdiler üzerinden etkinlik analizinin yapılması, performans açısından en kötü durumda bile algoritmanın nasıl çalıştığını gözlemlemeye imkan sağladığından, algoritmanın zaman karmaşıklığı için genellikle en-kötü durum analizi dikkate alınmaktadır.

3. FONKSİYONLARIN BÜYÜME HIZI

Yukarıda da ifade edildiği gibi, algoritmanın performansı ya da çalışma süresi ile algoritmanın girdi boyutu arasında doğrudan bir ilişki vardır. Girdi boyutu arttığında algoritmanın çalışma süresi de gözle görülür biçimde artmaktadır. Bununla birlikte girdi boyutu çok küçük olduğunda, çoğu algoritma için algoritmanın zaman karmaşıklığı da çok küçük olmaktadır. Dolayısıyla verilen bir algoritmanın etkinlik analizini yaparken çok küçük boyutlu girdiler kullanmak anlamlı

olmayacaktır. Yine benzer şekilde farklı algoritmaları etkinlikleri üzerinden kıyaslarken küçük boyutlu girdiler kullanıyor olmak ta sağlıklı sonuçlar üretmeyecektir.

Diğer yandan, algoritmaların etkinlik analiziyle asıl amaçlanan; algoritmaların tam olarak kaç adımda ya da tam olarak ne kadar sürede sonuç ürettiğini belirlemek değil, algoritmaların pratikte uygulanıp uygulanamayacağına dair bir fikir elde etmek ya da aynı problem için geliştirilmiş farklı algoritmaları performans açısından kıyaslayabilmektir. Bu tespitlerden hareketle; algoritmaların etkinlik analizinde yaptığımız aslında, yukarıda tanımlanan ve algoritmanın temel işlemlerinin algoritma tarafından kaç defa gerçekleştirildiğini bulan T(n) fonksiyonunun, küçük boyutlu girdiler için değil de büyük boyutlu girdiler için ne ürettiğini bulmak, veya en genel haliyle girdi boyutu n büyüdükçe T(n) fonksiyonunun nasıl değiştiğini anlamaya çalışmak, yani T(n) fonksiyonunun büyüme hızını belirlemek olacaktır. Örneğin $T(n) = 3n^3 + 2n^2 + 1$ fonksiyonunu göz önünde bulunduralım. $n = 10$ için $T(10) = 3.1000 + 2.100 + 1 = 3201$ olacaktır. Yine benzer şekilde $n = 100$ için $T(100) = 3.1000000 + 2.10000 + 1 = 3020001$ olacaktır. Görüleceği üzere n'nin değeri büyüdükçe fonksiyonun büyümesine en çok katkı yapan terim $3n^3$ terimidir. Hatta bu noktada fonksiyonun büyümesine katkısı çok az olduğundan terimin kaysayısı 3 göz ardı edilirse, fonksiyonun büyümesine en çok katkı yapan n^3 terimidir. Dolayısıyla $T(n) = 3n^3 + 2n^2 + 1$ fonksiyonunun büyüme hızı n^3'tür.

Özetleyecek olursak, algoritma etkinlik analizinde önce algoritmanın gerçekleştirdiği temel işlem ya da işlemler belirlenecek ve sonrasında istenen sonucu üretmek için bu işlemlerin algoritma tarafından kaç defa uygulandığını hesap eden T(n) fonksiyonu üretilecektir. Daha sonra ise, algoritmanın performansı için gösterge sayılabilecek T(n) fonksiyonunun büyüme hızı belirlenecektir. T(n) fonksiyonunun büyüme hızı bize algoritmanın pratikte uygulanıp uygulanamıyacağına ya da algoritmanın uygulanabilmesi için gerekli olan yatırıma değip değmeyeceğine dair fikir verecektir. Aşağıda küçükten büyüğe sıralı algoritma etkinlik analizinde yaygın olarak kullanılan fonksiyon büyüme hızları verilmiştir:

$$1 \quad \log n \quad n \quad n\log n \quad n^2 \quad n^3 \quad 2^n \quad 3^n \quad n!$$

Yukarıdaki skalaya göre örneğin verilen bir problem için geliştirilmiş iki farklı algoritmanın etkinlik analizi neticesinde T(n) fonksiyonlarının büyüme hızları sırasıyla log n ve n olarak belirlenmişse, birinci algoritmanın büyüme hızı daha düşük olduğundan, uygulamada performans açısından birinci algoritma tercih edilecektir. Algoritmaların etkinlik analizinde T(n) fonksiyonlarının büyüme hızları gözetilerek farklı algoritmaları kıyaslamak ve sıralamak için ya da geliştirilen bir algoritmanın yukarıdaki büyüme hızları skalasındaki uygun yerini tespit edebilmek için yaygın olarak 3 farklı asimptotik notasyon kullanılmaktadır. Bunlar büyük – Oh (O), büyük – Omega (W) ve büyük – Theta (Q) notasyonlarıdır.

4. ASIMPTOTIK NOTASYONLAR

Büyük – Oh (O) notasyonu asimptotik üst sınır olarak tanımlanmaktadır. Yani O(f(n)), f(n) fonksiyonu ile aynı ya da daha küçük büyüme hızına sahip fonksiyonları içeren bir küme tanımlamaktadır. Buna göre; örneğin $g(n) = 7n^2 + 2n - 5$ fonksiyonunun büyüme hızı n^2 olduğundan, $g(n) \in O(n^2)$'dir. Yine benzer şekilde, $h(n) = 3n + 2$ fonksiyonunun büyüme hızı n olduğundan ve n n^2'den küçük olduğundan, $h(n) \in O(n^2)$'dir. Fakat, $r(n) = 2n^4 + n$ fonksiyonunun büyüme hızı n^4 olduğundan ve n^4 n^2'den büyük olduğundan, $r(n) \notin O(n^2)$'dir.

Diğer yandan, büyük – Omega (Ω) notasyonu asimptotik alt sınır olarak tanımlanmaktadır. Yani $\Omega(f(n))$, f(n) fonksiyonu ile aynı ya da daha büyük büyüme hızına sahip fonksiyonları içeren bir küme tanımlamaktadır. Buna göre; örneğin $g(n) = 7n^2 + 2n - 5$ fonksiyonunun büyüme hızı n^2 olduğundan, $g(n) \in \Omega(n^2)$'dir. Yine benzer şekilde, $r(n) = 2n^4 + n$ fonksiyonunun büyüme hızı n^4 olduğundan ve n^4 n^2'den büyük olduğundan, $r(n) \in \Omega(n^2)$'dir. Fakat, $h(n) = 3n + 2$ fonksiyonunun büyüme hızı n olduğundan ve n n^2'den küçük olduğundan, $h(n) \notin \Omega(n^2)$'dir.

Son olarak, büyük – Theta (Θ) notasyonu asimptotik sıkı sınır olarak tanımlanmaktadır. Yani $\Theta(f(n))$, f(n) fonksiyonu ile tam olarak aynı büyüme hızına sahip fonksiyonları içeren bir küme tanımlamaktadır. Buna göre; örneğin $g(n) = 7n^2 + 2n - 5$ fonksiyonunun büyüme hızı n^2 olduğundan, $g(n) \in \Theta(n^2)$'dir. Fakat, $h(n) = 3n + 2$ fonksiyonunun büyüme hızı n olduğundan ve n n^2'den farklı olduğundan, $h(n) \notin O(n^2)$'dir. Yine benzer şekilde, $r(n) = 2n^4 + n$ fonksiyonunun büyüme hızı n^4 olduğundan ve n^4 n^2'den farklı olduğundan, $r(n) \notin O(n^2)$'dir.

Asimptotik notasyonlar için $g(n) \in O(f(n))$ (ya da $g(n) \notin O(f(n))$) gösterimi, $g(n) = O(f(n))$ (ya da $g(n) \neq O(f(n))$) şeklinde de kullanılabilmektedir. Ayrıca $g_1(n) = O(f_1(n))$ ve $g_2(n) = O(f_2(n))$ verildiğinde, $g_1(n) + g_2(n) = O(\max\{f_1(n), f_2(n)\})$ olacaktır. Yine benzer şekilde $g_1(n) = O(f_1(n))$ ve $g_2(n) = O(f_2(n))$ verildiğinde, $g_1(n).g_2(n) = O(f_1(n).f_2(n))$ olacaktır.

Yukarıda da ifade edildiği gibi; O(f(n)) f(n) fonksiyonu ile aynı ya da daha küçük büyüme hızına sahip fonksiyonları içeren bir küme tanımladığından, örneğin $g(n) = 3n^2 + n + 2$ fonksiyonu $O(4n^3)$ kümesinin bir elemanı olabileceği gibi, $O(2n^4)$ kümesinin de bir elemanı olabilmektedir. Yani $g(n) = 3n^2 + n + 2$ fonksiyonunun büyük – Oh gösterimi $g(n) = O(4n^3)$ olabileceği gibi, $g(n) = O(2n^4)$ de olabilmektedir. Bununla birlikte, $g(n) = 3n^2 + n + 2$ fonksiyonunun en sade ve en küçük büyük – Oh gösterimi $g(n) = O(n^2)$'dir. Dikkat edilirse, fonksiyonun en sade ve en küçük büyük – Oh gösterimi bize fonksiyonun büyüme hızını vermektedir.

Şimdi de $g(n) = (3n^3 + n)(2\log n + 1) + 5n^4$ fonksiyonunun en sade ve en küçük büyük – Oh gösterimini bulmaya çalışalım. Burada g(n) fonksiyonu, $g_1(n) = 3n^3 + n$, $g_2(n) = 2\log n + 1$ ve $g_3(n) = 5n^4$ olmak üzere $g(n) = g_1(n).\ g_2(n) + g_3(n)$ şeklinde yazılabilir. Önce $g_1(n)$, $g_2(n)$ ve $g_3(n)$ fonksiyonlarının büyük – Oh gösterimlerini bulalım. Görüleceği üzere, bu fonksiyonların en sade ve en küçük büyük – Oh gösterimleri sırasıyla $g_1(n) = O(n^3)$, $g_2(n) = O(\log n)$ ve $g_3(n) = O(n^4)$'tür. Yukarıda verilen $g_1(n) + g_2(n) = O(\max\{f_1(n), f_2(n)\})$ ve $g_1(n).g_2(n) = O(f_1(n).f_2(n))$ kurallarını kullanırsak g(n) fonksiyonundaki $g_1(n).\ g_2(n)$ teriminin büyük – Oh gösterimi $g_1(n).\ g_2(n) = O(n^3\log n)$ olacak ve dolayısıyla g(n) fonksiyonunun büyük – Oh gösterimi $g(n) = O(n^3\log n) + O(n^4) = O(n^4)$ olacaktır.

5. ÖZYİNELEMELİ OLMAYAN ALGORİTMALARIN ETKİNLİK ANALİZİ

Burada yukarıda tanımı verilen asimptotik büyük – Oh notasyonu kullanılarak, özyinelemeli olmayan algoritmaların zaman karmaşıklığının nasıl bulunduğu örnekler üzerinden anlatılacaktır. Algoritmaların zaman karmaşıklığı için en-kötü durum etkinlik analizi uygulanacaktır.

```
1   Buyuk[t1, t2, ..., tn])
2
3   girdi : n tamsayıdan oluşan bir dizi
4   çıktı : dizinin en büyük elemanı
5
6   temp ← t1
7   for (k = 2 to n)
8       if  tk > temp
9           temp ← tk
10  return temp
```

Şekil 7.2 Buyuk Algoritması Sözde Kodu

İlk olarak Şekil 7.2'de sözde kod ile gösterimi verilen ve verilen bir dizinin en büyük elemanını bulan algoritmayı göz önünde bulunduralım. Algoritma 'temp' isimli bir geçici değişken tanımlamakta ve dizinin ilk elemanını bu değişkene atamaktadır. Sonra **for** dögüsü yardımıyla sırasıyla ikinciden sonuncuya kadar dizinin elemanlarını 'temp' ile kıyaslamakta ve 'temp'in güncel değerinden daha büyük bir eleman keşfederse 'temp' değişkenine bu elemanı atamaktadır. Algoritma

döngünün sonunda 'temp' değişkeninin güncel değerine dönmektedir. Dikkat edilirse algoritmanın gerçekleştirdiği iki temel işlem vardır: kıyaslama ('t_i > temp') ve eğer 'temp'in güncel değerinden daha büyük bir sayı keşfedilirse atama ('temp ← t_i'). Burada en-kötü durum etkinlik analizi için göz önünde bulundurulması gereken girdi elemanları küçükten büyüğe sıralı bir dizidir. Çünkü böyle bir girdi için; dizinin bir sonraki elemanı her zaman için 'temp' değişkeninin güncel değerinden büyük olacağından, algoritma **for** döngüsünün her adımında hem kıyaslama hem de atama işlemini gerçekleştirecektir. **for** döngüsünün n – 1 adımı olduğundan, algoritma en-kötü durum için döngüde 2(n – 1) işlem gerçekleştirecektir. Algoritmanın başındaki 'temp' değişkenine başlangıç değerini atamayı da ayrı bir işlem olarak sayarsak T(n) = 2(n – 1) + 1 olacaktır. Hesaplanan T(n) = 2n – 1 fonksiyonunun en sade ve en küçük büyük – Oh gösterimi T(n) = O(n)'dir. Dolayısıyla Şekil 7.2'de verilen algoritmanın zaman karmaşıklığı O(n)'dir.

```
1  Adım(n)
2
3  temp ← 0
4  for (i = 1 to n)
5      for (k = i to n)
6          temp ← temp + 1
7  return temp
```

Şekil 7.3 Adım Algoritmasının Sözde Kodu

Şimdi de benzer şekilde Şekil 7.3'de sözde kod ile gösterimi verilen algoritmayı göz önünde bulunduralım. Algoritma 'temp' isimli bir geçici değişken tanımlamakta ve 0 sayısını bu değişkene atamaktadır. Algoritma iç içe **for** döngüsünün herbir adımında 'temp ← temp + 1' değişkeni üzerinden 'temp' değerini 1 artırmakta ve çıktı olarak 'temp' değişkeninin güncel değerine dönmektedir. Burada algoritmanın gerçekleştirdiği temel işlem atama işlemi 'temp ← temp + 1'dir. İçteki **for** döngüsünün i'den n'ye kadar (n – i + 1) adımı olduğundan, algoritma atama işlemini dıştaki **for** döngüsünün herbir adımında sırasıyla n, n – 1, n – 2, ..., 1 defa gerçekleştirmektedir. Dolayısıyla algoritmanın başındaki 'temp' değişkenine başlangıç değerini atamayı da ayrı bir işlem olarak sayarsak T(n) = n + n – 1 + n – 2 + ... + 1 + 1 = $(n^2 + n + 2) / 2$ olacaktır. Hesaplanan T(n) = $(n^2 + n + 2) / 2$ fonksiyonunun en sade ve en küçük büyük – Oh gösterimi T(n) = O(n^2)'dir. Dolayısıyla Şekil 7.3'de verilen algoritmanın zaman karmaşıklığı O(n^2)'dir.

```
1  Simetrik([t1,1, ...,t1,n;...; tn,1, ...,tn,n])
2
3  girdi : nxn boyutlu bir matris
4  çıktı : matris simetrikse true; değilse false
5
6  for (i = 1 to n)
7      for (k = 1 to n
8          if t[i][k] ≠ t[k][i]
9              return false
10 return true
```

Şekil 7.4 Simetrik Algoritmasının Sözde Kodu

Son olarak Şekil 7.4'de sözde kod ile gösterimi verilen ve verilen bir matrisin simetrik olup olmadığını belirleyen algoritmayı göz önünde bulunduralım. Görüleceği üzere algoritmanın temel işlemi kıyaslamadır ('t[i][k] ≠ t[k][i]'). Dikkat edilirse, dıştaki **for** döngüsünde n adım olduğundan, ve döngünün her adımı için içteki **for** döngüsünde yine n defa kıyaslama gerçek-

leştirildiğinden, algoritma en-kötü durum için n^2 kıyaslama gerçekleştirecek ve dolayısıyla T(n) = n^2 olacaktır. Burada en-kötü durum etkinlik analizi için göz önünde bulundurulması gereken girdi simetrik bir matristir. Hesaplanan T(n) = n^2 fonksiyonunun en sade ve en küçük büyük – Oh gösterimi T(n) = O(n^2)'dir. Dolayısıyla Şekil 7.4'de verilen algoritmanın zaman karmaşıklığı O(n^2)'dir.

```
1  Kuvvet(X, N)
2
3  girdi : X tamsayısı ve N pozitif tamsayısı
4  çıktı : X'in N. kuvveti
5
6  if N = 0
7      return 1
8  else
9      return X.Kuvvet(X, N – 1)
```

Şekil 7.5 Kuvvet Algoritmasının Sözde Kodu

6. ÖZYINELEMELI ALGORITMALARIN ETKINLIK ANALIZI

Burada yukarıda tanımı verilen asimptotik büyük – Oh notasyonu kullanılarak, özyinelemeli algoritmaların zaman karmaşıklığının nasıl bulunduğu örnekler üzerinden anlatılacaktır. Özyinelemeli algoritmalar için de, özyinelemeli olmayan algoritmaların etkinlik analizindekine benzer olarak; önce algoritmanın temel işlemleri belirlenecek ve algoritmanın bu temel işlemleri kaç defa uyguladığını hesaplayan T(n) fonksiyonu üretilecektir. Yalnız özyinelemeli algoritmalar için T(n) fonksiyonunu üretmek kolay olmamaktadır. Önce T(n) fonksiyonu için özyineleme bağıntısının belirlenmesi ve sonrasında bu bağıntının çözülmesi gerekmektedir. T(n) fonksiyonu için özyineleme bağıntısı belirlendikten ve çözüldükten sonra, algoritmanın zaman karmaşıklığı için yine yukarıdaki analizlere benzer T(n) fonksiyonunun en sade ve en küçük büyük – Oh gösterimi bulunacaktır.

İlk olarak Şekil 7.5'de sözde kod ile gösterimi verilen ve verilen bir X tamsayısının N. kuvvetini bulan özyinelemeli algoritmayı göz önünde bulunduralım. Algoritma 'N = 0' koşulu sağlanırsa 1 sayısına dönmekte, aksi halde 'X.Kuvvet(X, N – 1)' komutuyla girdi boyutu 1 azaltılarak algoritma tekrar çalıştırılmaktadır. Algoritmanın gerçekleştirdiği temel işlemi kıyaslama olarak belirtirsek, algoritma içerisinde bir temel işlem gerçekleştirilmekte ve girdi boyutu bir azaltılarak algoritma tekrar çalıştırılmaktadır. Algoritmanın zaman karmaşıklığına T(n) dersek, algoritmanın içerisindeki özyineleme çağrısının (yani 'X.Kuvvet(X, N – 1)' komutunun) zaman karmaşıklığı T(n – 1) olacağından, T(n) = T(n – 1) + 1 olacaktır. Girdi boyutu N = 0 olduğunda, algoritma tek işlem gerçekleştirerek 0 değerine döneceğinden T(0) = 1'dir. Şimdi T(0) = 1 başlangıç koşulunu göz önünde bulundurarak T(n) = T(n – 1) + 1 bağıntısını çözelim. Burada bağıntıyı çözmekten kasıt, eşitliğin sağ tarafının belli bir takım yöntemler kullanarak T fonksiyonuna bağımlılığını ortadan kaldırmak ve eşitliğin sağında sadece n değişkeninden oluşan ifadeler elde etmektir.

$$T(n) = T(n-1) + 1 \qquad (1)$$

ifadesinde n gördüğümüz yere n – 1 yazarsak T(n – 1)'in karşılığı T(n – 1) = T(n – 2) + 1 bulunacaktır. Bunu ilk bağıntıda eşitliğin sağındaki T(n – 1)'in yerine yazarsak,

$$T(n) = T(n-1) + 1 = T(n-2) + 1 + 1 \qquad (2)$$

elde etmiş oluruz. Yine benzer şekilde (1) eşitliğinde n gördüğümüz yere n – 2 yazarsak T(n –

2)'in karşılığı T(n – 2) = T(n – 3) + 1 bulunacaktır. Bunu (2) bağıntısında eşitliğin sağındaki T(n – 2)'in yerine yazarsak,

$$T(n) = T(n-2) + 1 + 1 = T(n-3) + 1 + 1 + 1 \qquad (3)$$

elde etmiş oluruz. Bu şekilde devam edilirse i. adımda

$$T(n) = T(n-i) + i.1 \qquad (4)$$

DİKKAT EDELİM

Özyinelemi algoritmaların etkinlik analizinde, önce temel işlemlerin algoritma tarafından kaç defa gerçekleştirildiğini hesap eden T(n) fonksiyonu için özyineleme bağıntısı belirlenmeli ve sonrasında zaman karmaşıklığı için bu bağıntı çözülmelidir.

ifadesi elde edilecektir. Burada i = n için (4) eşitliğinde T(n) = T(0) + n elde edilecektir. T(0) = 1 olduğundan, (4) eşitliği T(n) = n + 1 olacaktır. Hesaplanan T(n) = n + 1 fonksiyonunun en sade ve en küçük büyük – Oh gösterimi T(n) = O(n)'dir. Dolayısıyla Şekil 7.5'de verilen algoritmanın zaman karmaşıklığı O(n)'dir.

```
1   EnBuyuk([t_i, t_i+1, ..., t_j])
2
3   girdi : N elemanlı bir tam sayı dizisi
4   çıktı : Dizinin en büyük elemanı
5
6   // dizide tek bir eleman kaldıysa algoritma o elemana döner
7   if i = j
8       return t_i
9   else
10      mid ← (i + j)/2
11      // dizinin ilk yarısının en küçük elemanı
12      buyuk1 ← EnBuyuk([t_i, ..., t_mid])
13      // dizinin ikinci yarısının en küçük elemanı
14      buyuk2 ← EnBuyuk([t_mid+1, ..., t_j])
15  if buyuk1 < buyuk2
16      return buyuk2
17  else
18      return buyuk1
```

Şekil 7.6 EnBuyuk Algoritmasının Sözde Kodu

Şimdi de Şekil 7.6'da sözde kod ile gösterimi verilen ve verilen bir tamsayı dizisinin en büyük elemanını bulan özyinelemeli algoritmayı göz önünde bulunduralım. 'i = j' koşulu sağlanırsa algoritma t_i sayısına dönmekte, aksi halde 'EnBuyuk($[t_i,...,t_{mid}]$)' ve 'EnBuyuk($[t_{mid+1},...,t_j]$)' komutlarıyla girdi boyutu yarıya indirilerek algoritma tekrar çalıştırılmaktadır. Dikkat edilirse 'mid ← (i + j)/2' atamasını ve 'buyuk1 < buyuk2' kıyaslamasını da algoritmanın gerçekleştirdiği temel işlemlerden sayarsak, algoritma içerisinde 3 temel işlem gerçekleştirilmektedir. Ayrıca 'EnBuyuk($[t_i,...,t_{mid}]$)' ve 'EnBuyuk($[t_{mid+1},...,t_j]$)' komutlarıyla algoritma iki defa daha çalıştırılmaktadır. Algoritmanın zaman karmaşıklığına T(n) dersek, algoritmanın içerisindeki özyineleme çağrılarının (yani 'EnBuyuk($[t_i,...,t_{mid}]$)' ve 'EnBuyuk($[t_{mid+1},...,t_j]$)' komutlarının) zaman karmaşıklı-

ğı T(n/2) olacağından, T(n) = 2T(n/2) + 3 olacaktır. Girdi boyutu N = 1 olduğunda, algoritma tek işlem gerçekleştirerek 't_i' değerine döneceğinden T(1) = 1'dir. Şimdi T(1) = 1 başlangıç koşulunu göz önünde bulundurarak T(n) = 2T(n/2) + 3 bağıntısını çözelim.Yukarıda çözüme benzer şekilde,

$$T(n) = 2T(n/2) + 3 \qquad (7)$$

ifadesinde n gördüğümüz yere n/2 yazarsak T(n/2)'nin karşılığı T(n/2) = 2T(n/4) + 3 bulunacaktır. Bunu ilk bağıntıda eşitliğin sağındaki T(n/2)'nin yerine yazarsak,

$$T(n) = 2T(n/2) + 3 = 2(2T(n/4) + 3) + 3 = 4T(n/4) + 3(1 + 2) \qquad (8)$$

elde etmiş oluruz. Yine benzer şekilde (7) ifadesinde n gördüğümüz yere n/4 yazarsak T(n/4)'ün karşılığı T(n/4) = 2T(n/8) + 3 bulunacaktır. Bunu (8) bağıntısında eşitliğin sağındaki T(n/4)'ün yerine yazarsak,

$$T(n) = 4(2T(n/8) + 3) + 3(1 + 2) = 8T(n/8) + 3(1 + 2 + 4) \qquad (9)$$

elde etmiş oluruz. Bu şekilde devam edilirse i. adımda

$$T(n) = 2^i\, T(n/2^i) + 3(1 + 2 + \ldots + 2^{i-1}) \qquad (10)$$

ifadesi elde edilecektir. Burada i = logn için $T(n) = 2^{\log n}T(1) + 3(1 + 2 + \ldots + 2^{\log n - 1}$ olacaktır. Eşitliğin sağını düzenlersek, T(1) = 1 olduğundan T(n) = 4n olacaktır. Hesaplanan T(n) = 4n fonksiyonunun en sade ve en küçük büyük – Oh gösterimi T(n) = O(n)'dir. Dolayısıyla Şekil 7.6'da verilen algoritmanın zaman karmaşıklığı O(n)'dir.

BÖLÜM ÖZETİ

Bu bölümde algoritmanın verilen girdi için ne kadar sürede ya da ne kadar adımda istenen çıktıyı ürettiğinin belirlenmesi olarak tanımlayabileceğimiz algoritmanın zaman karmaşıklığının nasıl hesaplandığı örnekler üzerinden detaylıca incelenmiştir. Bir algoritmanın verilen girdi için ne kadar sürede ya da ne kadar adımda istenen çıktıyı ürettiğini ölçerken; ilk yapılması gereken bu ölçümün hangi birim kullanılarak yapılacağının belirlenmesidir. Her ne kadar akla ilk gelen zaman karmaşıklığını saniye ya da dakika gibi zaman birimleri cinsinden ölçmek olsa da bu noktada bilgisayarın sahip olduğu donanımın kalitesi, algoritmanın hangi programlama diline nasıl aktarıldığı ya da programlama dili için hangi derleyicinin kullanıldığı gibi etkenler algoritmanın çalışma süresini yani zaman karmaşıklığını doğrudan etkilemektedirler. Algoritmaların zaman karmaşıklığını daha sağlıklı bir şekilde hesaplayabilmek için etkinlik analizinin, algoritmaların üzerinde çalıştırılacağı bilgisayarın sahip olduğu donanımın kalitesi, algoritmanın hangi programlama diline nasıl aktarıldığı gibi etkenlerden bağımsız bir şekilde yapılması daha doğru bir yöntem olacaktır. Algoritmanın zaman karmaşıklığını ölçerken uygulayabileceğimiz yöntem, algoritmanın istenen çıktıyı üretirken gerçekleştirdiği temel işlemleri belirlemek ve bu işlemlerin algoritma sonlandırılıncaya kadar kaç defa gerçekleştirildiğini hesaplayan bir T(n) fonksiyonu tanımlamak olacaktır. Bu fonksiyon algoritmanın girdi boyutu üzerinde çalışacaktır. Sonrasında, algoritmanın performansı için gösterge sayılabilecek T(n) fonksiyonunun büyüme hızı belirlenecektir. T(n) fonksiyonunun büyüme hızı bize algoritmanın pratikte uygulanıp uygulanamayacağına ya da algoritmanın uygulanabilmesi için gerekli olan yatırıma değip değmeyeceğine dair fikir verecektir. Bu bölümde ayrıca etkinlik analizinde T(n) fonksiyonlarının büyüme hızları gözetilerek farklı algoritmaları kıyaslamak ve sıralamak için ya da geliştirilen bir algoritmanın büyüme hızları skalasındaki uygun yerini tespit edebilmek için yaygın olarak kullanılan büyük – Oh (O), büyük – Omega () ve büyük – Theta (Θ) asimptotik notasyonları incelenmiştir. Diğer yandan asimptotik büyük – Oh notasyonu kullanılarak, özyinelemeli ya da özyinelemeli olmayan algoritmaların zaman karmaşıklığının nasıl bulunduğu da örnekler üzerinden anlatılmıştır.

GÖZDEN GEÇİRELİM

1. Toplam([t_1,...., t_n])
temp ← 0
for (k = 1 to n){
temp ← temp + t_k}
return temp

T(n) fonksiyonu algoritmanın temel işlemi ya da işlemlerinin algoritma tarafından kaç defa gerçekleştirildiğini sayan fonksiyon olarak tanımlanmaktadır. Buna göre yukarıda sözde kod ile gösterimi verilen algoritmanın T(n) fonksiyonunun kuralı aşağıdakilerden hangisinde doğru olarak verilmiştir?

a) $T(n) = 2n^2 + 2$
b) $T(n) = n^2 + 2$
c) $T(n) = 2n + 1$
d) $T(n) = n - 3$
e) $T(n) = \log n + 3$

2. İkiliArama(A,i,k,x)
if (i >= k)
return false
mid ← (i+ k)/2
if (A[mid] = x)
return true
elseif (A[mid] > x)
return İkiliArama(A,i,mid-1,x)
else
return İkiliArama(A,mid+1,k,x)

Yukarıda sözde kod ile gösterimi verilen özyinelemeli algoritmanın zaman karmaşıklığının özyineleme bağıntısı aşağıdakilerden hangisinde doğru olarak verilmiştir?

a) $T(n) = T(n/2) + n$
b) $T(n) = T(n/2) + 4$
c) $T(n) = 2T(n/2) + 1$
d) $T(n) = 3T(n/2) + 2$
e) $T(n) = 4T(n/2) + 3$

3. Büyük – Omega (Ω) notasyonu asimptotik alt sınır olarak tanımlanmaktadır. Buna göre $f(n) = 2n^3+1$ fonksiyonu için aşağıdakilerden hangisi doğru değildir?

a) $f(n) = \Omega(\log n)$
b) $f(n) = \Omega(n)$
c) $f(n) = \Omega(n^2)$
d) $f(n) = \Omega(n^3)$
e) $f(n) = \Omega(n^4)$

4. Büyük – Theta (Θ) notasyonu asimptotik sıkı sınır olarak tanımlanmaktadır. Buna göre $f(n) = n^2+3$ fonksiyonu için aşağıdakilerden hangisi doğrudur?

a) $f(n) = \Theta(\log n)$
b) $f(n) = \Theta(n)$
c) $f(n) = \Theta(n^2)$
d) $f(n) = \Theta(n^3)$
e) $f(n) = \Theta(n^4)$

5. Büyük – Oh (O) notasyonu asimptotik üst sınır olarak tanımlanmaktadır. Buna göre $f(n) = 2n^3+1$ fonksiyonu için aşağıdakilerden hangisi doğru değildir?

a) $f(n) = O(n)$
b) $f(n) = O(n^3)$
c) $f(n) = O(n^3 \log n)$
d) $f(n) = O(n^4)$
e) $f(n) = O(n^5)$

6. Kuvvet(X,n)
```
temp ← 1
for ( k = 1 to n){
    temp ← temp.x}
return temp
```

Yukarıda sözde kod ile gösterimi verilen algoritmanın zaman karmaşıklığı aşağıdakilerden hangisinde doğru olarak verilmiştir?

a) O(logn)

b) O(n)

c) O(nlogn)

d) $O(n^2)$

e) $O(n^3)$

7. Toplam([t1,..., tn])
```
temp1 ← 0
for ( i = 1 to n){
    for ( j = 1 to n){
        temp2 ← 0
        for ( k = 1 to n){
            temp2 ← temp2 + k
            temp1 ← i.temp1 – j.temp2}}}
return temp1
```

Yukarıda sözde kod ile gösterimi verilen algoritmanın zaman karmaşıklığı aşağıdakilerden hangisinde doğru olarak verilmiştir?

a) O(logn)

b) O(n)

c) O(nlogn)

d) $O(n^2)$

e) $O(n^3)$

8. EnBuyuk($[t_1,..., t_n]$)
```
temp ← t1
for ( k = 2 to n){
    if (tk > temp){
        temp ← tk}}
return temp
```

Yukarıda sözde kod ile gösterimi verilen EnBuyuk algoritmasının farklı elemanlardan oluşan bir sayı dizisini girdi olarak aldığı kabul edilirse en-iyi durum zaman karmaşıklığını verecek olan girdi aşağıdakilerden hangisidir?

a) Girdinin ilk elemanının en büyük eleman olması

b) Girdinin son elemanının en büyük eleman olması

c) Girdinin küçükten büyüğe sıralı elemanlardan oluşması

d) Girdinin ortanca elemanının en büyük eleman olması

e) Girdinin ilk yarısının küçükten büyüğe ikinci yarısının büyükten küçüğe elemanlardan oluşması

9. **Büyük – Oh (O) notasyonu asimptotik üst sınır olarak tanımlanmaktadır. Buna göre $f(n) = n^7+3$ fonksiyonu için aşağıdakilerden hangisi doğrudur?**

a) $f(n) = O(n)$

b) $f(n) = O(n^2)$

c) $f(n) = O(n^4)$

d) $f(n) = O(n^6)$

e) $f(n) = O(n^8)$

10. Büyük – Omega (Ω) notasyonu asimptotik alt sınır olarak tanımlanmaktadır. Buna göre f(n) = n^2+3 fonksiyonu için aşağıdakilerden hangisi doğrudur?

a) f(n) = Ω(n)

b) f(n) = Ω(n^3)

c) f(n) = Ω(n^5)

d) f(n) = Ω(n^7)

e) f(n) = Ω(n^9)

Yanıt Anahtarı: 1-C 2-B 3-E 4-C 5-A 6-B 7-E 8-A 9-E 10-A

Kaynakça

Levitin A. (2012) Introduction to The Design & Analysis of Algorithms, 3rd Edition, Pearson.

Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. (2009) Introduction to Algorithms, 3rd Edition, MIT Press.