Zaman ve Hafıza Karmaşıklığı Analizi

*Big O Notation in C

1st Muhammet Cüneyd Kurtbaş
Kocaeli Üniversitesi
Bilgisayar Mühendisliği
200201132
cuneydkurtbas@gmail.com

***Abstract*—Karmaşıklık analizi, bir algoritmanın çalışması için gereken kaynak miktarının belirlenmesidir. Yani algoritmanın performansını ve kaynak kullanımını ölçen teorik bir çalışmadır. Bilgisayar biliminde belli bir probleme birden fazla çözüm yolu üretmek mümkündür. Tabi ki burada en uygun çözümü, yani algoritmayı seçmek önemlidir. Bir problemi çözmeye çalışırken olası tüm algoritmaları incelemeli ve en uygun olanı seçmemiz gerekiyor. Algoritma bir şekilde tasarlanır ve problem çözülür. Ama her bakımdan en iyi sonucu veren tek bir algoritma vardır. Karmaşıklık analizinin amacı da o en uygun sonucu veren algoritmayı bulmaktır.**

***Index Terms*—BigO, time complexity, space complexity**

I. GİRİŞ

Zaman karmaşıklığı (Time Complexity), algoritmanın yürütme zamanının derecesinin asimptotik notasyonlarla gösterilmesidir. Bir başka deyişle bir algoritmanın çalışması için gereken süredir. Bu süre ne kadar kısa olursa algoritmanın performansı o kadar iyi olur. Zaman karmaşıklığını bir algoritmanın performansı olarak düşünebiliriz.

Bu süre milisaniyeleri hesaplayarak değil, yapılan işlem sayısına göre hesaplanmaktadır. Tabi ki bu hesaplama yapılırken algoritmadaki veri setinin büyüklüğüne ve sırasına dikkat edilir. Amaç, giriş boyutu ve zaman arasındaki karmaşıklığı azaltmaktır. Yani detayları es geçerek çalışma süresini indirgemektir.

İşte bu yüzden algoritma sonsuza giderken asimptotik gösterimde sabitler ve katsayılar gibi büyümeye fazla etkisi olmayan kısımlar hesaplanmaz. Böylece geriye algoritmanın büyümesinde asıl etkiye sahip olan değerler kalır ve gerçek fonksiyona göre yaklaşık bir değer hesaplanır. Bir bilgisayarın donanım özellikleri ne kadar iyi olursa olsun, her zaman en iyi performans gösteren algoritmayı seçmek gerekir. Böylece donanımda yapılan iyileştirmelerle algoritma daha da iyi performans göstermiş olur.

Alan karmaşıklığı (Space Complexity), bir algoritmanın bir girdi boyutu için çıktı üretirken ihtiyaç duyduğu bellek miktarıdır. Yani buna alan maliyetini ölçmek demektir diyebiliriz. Daha iyi alan karmaşıklığına sahip olan algoritma bellekten daha az alan tüketir. Algoritmanın performansını artırmak için sistem belleğini artırmak ilk akla gelen çözüm olsa da, her zaman daha az alan tüketen bir algoritma tasarlamak daha mantıklıdır. Alan karmaşıklığı, algoritmayı çalıştıran sistem, derleyici ve programlama dili gibi çeşitli faktörlere bağlıdır. Bu yüzden algoritmanın kendisini analiz ederken bu faktörleri dikkate almamalıyız.

Ne yazık ki, algoritma performansında zaman ve alan verimliliği birbirine terstir. Yani hız arttıkça bellek tüketimi artar. Ya da tam tersi.Örnek olarak merge sort, bubble sort ve in-place heap algoritmalarını inceleyelim. Merge sort en hızlı ancak en fazla bellek tüketen algoritmadır. Bubble sort ise tam tersine en yavaş ancak en az bellek tüketen algoritmadır. In-place heap ise bu iki algoritmanın arasında performans gösteren daha dengeli bir algoritmadır.[1]

II. YÖNTEM

Notation, karmaşıklık analizi yapılan bir algoritmanın performansının farklı gösterimler ile temsil edilmesidir. Zaman karmaşıklığında genel olarak üç temel asimptotik notasyon vardır. Bunlar: Big-O Notasyonu, Omega Notasyonu, Teta Notasyonudur.

A. Omega Notasyonu

Omega notasyonu (Omega Notation), asimptotik alt sınırı(lower bound) ifade eder. Bu notasyonla algoritma zaman açısından en iyi duruma(best case) ulaşmış olur. Büyük-O gösteriminin tersidir.

B. Teta Notasyonu

Teta gösterimi, alt sınır(lower bound) ile üst sınır(upper bound) arasında kalan ortalama bir karmaşıklığı ifade eder.. Yani, Big-O notasyonu ile Big Omega notasyonu arasında yer alır.

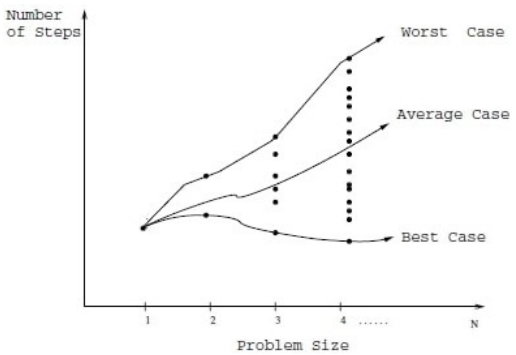

Fig. 1. Algoritmalarda best case, worst case ve average case [1]

C. Big-O Notasyonu

Big O Notasyonu, algoritma analizindeki zaman karmaşıklığında üst sınırı(upper bound) tanımlar. Algoritmanın büyüme hızını (growth rate) temsil etmek için kullanılır. Bir algotirmanın karmaşıklık analizi yapıldıktan sonra ortaya çıkan ifadeleri sadeleştirerek algoritmanın en sade karmaşıklığını bulmamızı sağlar. Big-O kavramının ne olduğunu bilmeden algoritma tasarlamak geliştirdiğiniz algoritmaya zararlar verebilir.

Algoritmaları tasarlarken farklı veri yapılarına ihtiyaç duyarız. Array, LinkedList, Queue, Stack, Binary Tree, Graph gibi. Ya da aynı veri yapısı üzerinde (örneğin Array) farklı algoritmalar çalıştırabiliriz. Bubble Sort, Insertion Sort, Quick Sort, Heap Sort, Merge Sort sıralama algortimaları buna örnektir. Seçtiğimiz veri yapıları algoritmanın kalitesine, tükettiği kaynaklara, hızına ve kullanışına farklı etkiler yapar. Bu yüzden yazılım geliştiricilerin bu konuyu iyi bir şekilde anlaması gerekmektedir.

En yaygın çalışma zamanları O(log n), O(nlog n), O(n), O(n²)' dir. Ancak çalışma zamanlarının kesin bir listesi yoktur. Çünkü çalışma zamanı hesaplanırken birçok değişken göz önüne alınır.

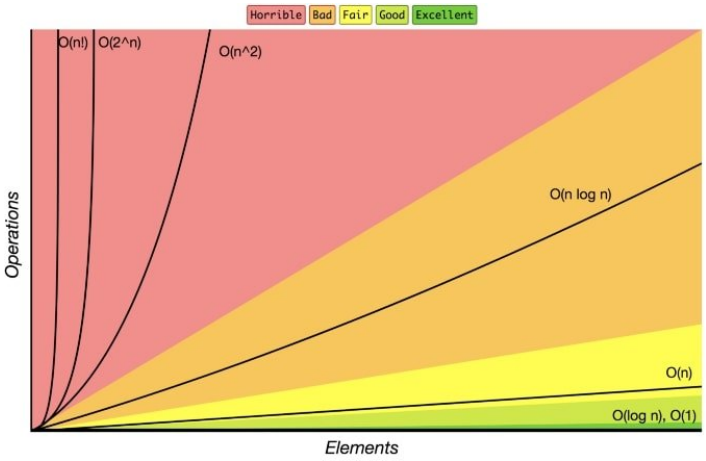

Fig. 2. Big-O Karmaşıklık Grafiği[1]

1) Sabit Karmaşıklık (Constant Complexity) – O(1): Sabit karmaşıklıkta algoritmaya girilen veri seti ne kadar büyük olursa olsun çalışma zamanı ve kullanılan kaynak miktarı sabittir. O(1) zamanda çalışır. Örneğin;

```
var arr = [ 1,2,3,4,5];
arr[3]; // => 4
```

Algoritmaya girdi olarak girilen dizinin boyutu ne olursa olsun çalışma zamanı hep sabit kalacaktır. Çünkü biz dizinin 3. elemanını buluyoruz.

2) Doğrusal Karmaşıklık (Linear Complexity) – O(N): Doğrusal karmaşıklıkta algoritmaya girdiğimiz veri setinin büyüklüğü arttıkça çalışma zamanı da doğrusal olarak artar. Yani girilen veri setinin büyüklüğü ile çalışma zamanı arasında doğru orantı vardır. O(N) zamanda çalışır. Örneğin;

```
for(int i=0; i<n; i++){
    print("Hello World");
}
```

çıktı olarak n tane "Hello World" yazacaktır. Dolayısıyla n sayısı arttıkça programın çalışma zamanı da doğrusal olarak artacaktır.

3) Quadratic Complexity – O(N²): Basitçe, çalışma zamanı girdi büyüklüğünün karesiyle doğru orantılıdır. Yani eğer girdi büyüklüğü 2 ise 4 işlem, 8 ise 16 işlem gerçekleşir. O(N²) zamanda çalışır.Genellikle sıralama(sorting) algoritmalarında bu karmaşıklık görülür.

```
for(int i=0; i<arr.length; i++)
    for(int j=0; j<arr.length; j++)
        print("Hello World");
```

Tek bir for döngüsünde çalışma zamanı O(N) olur. Ancak iç içe 2 tane for döngüsü kullanıldığı zaman çalışma zamanı O(N²)'dir.

4) Logaritmik Karmaşıklık (Logarithmic Complexity) – O(log N): Logaritmik karmaşıklık genel olarak her seferinde problemi ikiye bölen algoritmalarda karşımıza çıkar. Diğer bir deyişle, algoritmayı çalıştırmak için geçen süre N girdi boyutunun logaritması ile orantılıysa bu algoritma logaritmik zaman karmaşıklığına sahiptir. O(logN) zamanda çalışır.

```
for(let i=n; i>1; i/=2){
    console.log(i);
}
```

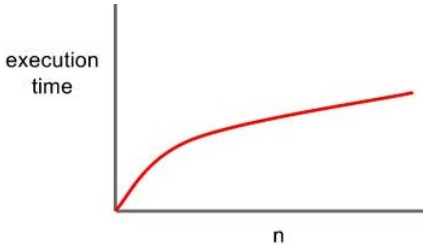

Fig. 3. Logaritmik Karmaşıklık

5) Linearitmik Complexity – O(NlogN): Linearitmik zaman karmaşıklığı, doğrusal bir algoritmadan biraz daha yavaştır, ancak yine de ikinci dereceden bir algoritmadan çok daha iyidir. O(NlogN) zamanda çalışır.

6) Üstel Karmaşıklık (Exponential Complexity) – O(2^N) : Üstel karmaşıklık (2 tabanında) logaritmanın tersi şekilde işlem sayısı üstel olarak algoritmalarda görülür. İşlem açısından katlanarak artan bu tip fonksiyonlar sisteminizi tüketir. Bu tip hesaplamalardan olabildiğince kaçınmak gerekir. O(2^N) zamanda çalışır. Üstel karmaşıklığa örnek olarak bir dizinin tüm alt kümelerini verebiliriz.

7) Faktöriyel Karmaşıklığı (Factorial Complexity) – O(N!): Faktöriyel, kendisinden küçük tüm pozitif tam sayıların çarpımıdır. Dolayısıyla faktöriyel karmaşıklığa sahip algoritmaların karmaşıklığı oldukça hızlı büyür. Bu yüzden bu karmaşıklığa sahip algoritmalardan uzak durmak gerekir. O(N!) zamanda çalışır.

III. SÖZDE KOD

```
void YerKarmasikligiHesapla
(char *trimSatir, int yerKarmasikligi[]){
size_t n =
sizeof(degiskenTipleriByteDegerleri)/
sizeof(degiskenTipleriByteDegerleri[0]);
char *temp = strstr(trimSatir, "[");

for(int i=0;i<n;i++){
if(StartsWith(degiskenTipleri[i],
trimSatir) == 1){
if(temp == NULL){
char parcalanacakKisim[] = "";
SubString(parcalanacakKisim, trimSatir,
strlen(degiskenTipleri[i]),
strlen(trimSatir)-strlen
(degiskenTipleri[i]));
int degiskenSayisi = 0;
char *parca =
strtok(parcalanacakKisim,",");
while(parca != NULL){
degiskenSayisi++;
parca = strtok(NULL , ",");
}
yerKarmasikligi[0] += (degiskenSayisi *
degiskenTipleriByteDegerleri[i]);
break;
}else{
char *tempSatir = trimSatir;
int boyut = 0, hesap = 0;
char *parca = strtok(tempSatir , "[");
parca = strtok(NULL , "[");//n]
char sayi[] = "";

while(parca != NULL){
if(isdigit(parca[0])){
for(int i=0; parca[i] != ']';i++){
sprintf(sayi, "%s%c", sayi,parca[i]);
}
//printf("%s",sayi);
hesap += (atoi(sayi) *
degiskenTipleriByteDegerleri[i]);
}else{
        boyut++;
            }
parca = strtok(NULL , "[");
        }
if(hesap == 0)
hesap = degiskenTipleriByteDegerleri[i];
yerKarmasikligi[boyut] += hesap;
break;
            }
        }

int ForNMi(char *satir){
char *tempSatir = satir;
if(StartsWith("for(", satir) == 1){
SubString(tempSatir, satir, 4, strlen(satir)-5);
}else if(StartsWith("for (", satir) == 1){
SubString(tempSatir, satir, 5, strlen(satir)-6);
 }else
 return 0;

char *tanimKismi = strtok(tempSatir,";");
char *kontrolKismi = strtok(NULL, ";");
char *artirmaKismi = strtok(NULL, ";");

kontrolKismi = Trim(kontrolKismi);

char *temp1 = strstr(artirmaKismi, "/");
char *temp2 = strstr(artirmaKismi, "*");
if(temp1 != NULL || temp2 != NULL)
return -2;

char *sagTaraf = strtok(kontrolKismi, "=<>!");
sagTaraf = strtok(NULL, "=<>!");
if(isdigit(sagTaraf[0]) == 1)
return atoi(sagTaraf);
return -1;
}
int WhileNMi(char *satir){
char *tempSatir = satir;
if(StartsWith("while(", satir) == 1){
SubString(tempSatir, satir, 6, strlen(satir)-7);
}else if(StartsWith("while (", satir) == 1){
SubString(tempSatir, satir, 7, strlen(satir)-8);
}else if(StartsWith("}while(", satir) == 1){
SubString(tempSatir, satir, 7, strlen(satir)-8);
}else if(StartsWith("}while (", satir) == 1){
SubString(tempSatir, satir, 8, strlen(satir)-9);
}else if(StartsWith("} while (", satir) == 1){
SubString(tempSatir, satir, 9, strlen(satir)-10);
}else if(StartsWith("} while(", satir) == 1){
SubString(tempSatir, satir, 8, strlen(satir)-9);
}else
return;

char *solTaraf = strtok(tempSatir, "=<>!");
char *sagTaraf = strtok(NULL, "=<>!");

solTaraf = Trim(solTaraf);
sagTaraf = Trim(sagTaraf);

if(sagTaraf == NULL){
return -1;
}

if(isdigit(sagTaraf[0]) == 1)
return atoi(sagTaraf);
return -1;
}
```

IV. ÇIKTILAR

```
Asagidakilerden secim yapiniz:
---------------------------------------
        0 : Cikis
        1 : Karmasiklik Hesapla
        2 : Calisma Suresi Hesapla
        3 : Konsolu Temizle
---------------------------------------
Secim : 1
Karmasiklik hesaplanacak dosyanin adini

        Yer Karmasikligi = (4n + 12)

        Zaman Karmasikligi = O(n)
```

Fig. 4. Karmaşıklık Hesaplama

```
Asagidakilerden secim yapiniz:
-----------------------------------------------------
       0 : Cikis
       1 : Karmasiklik Hesapla
       2 : Calisma Suresi Hesapla
       3 : Konsolu Temizle
-----------------------------------------------------
Secim : 2
Dosya adini (uzantisi olmadan) giriniz: k3

720     Verilen kodun calisma suresi 0.068000 saniye.
```

Fig. 5. Çalışma Süresi Hesaplama

V. SONUÇ

Bilgisayar bilimlerinde bir algoritmanın incelenmesi sırasında sıkça kullanılan bu terim çalışmakta olan algoritmanın en kötü ihtimalle ne kadar başarılı olacağını incelemeye yarar. Bilindiği üzere bilgisayar bilimlerinde yargılamalar kesin ve net olmak zorundadır. Tahmini ve belirsiz karar verilmesi istenmeyen bir durumdur. Bir algoritmanın ne kadar başarılı olacağının belirlenmesi de bu kararların daha kesin olmasını sağlar. Algoritmanın başarısını ise çalıştığı en iyi duruma göre ölçmek yanıltıcı olabilir çünkü her zaman en iyi durumla karşılaşılmaz.Algoritma analizinde kullanılan en önemli iki ölçü hafıza ve zaman kavramlarıdır. Yani bir algoritmanın ne kadar hızlı çalıştığı ve çalışırken ne kadar hafıza ihtiyacı olduğu, bu algoritmanın performansını belirleyen iki farklı boyuttur. En iyi algoritma en hızlı ve en az hafıza ihtiyacı ile çalışan algoritmadır. İşte en kötü durum analizi olayın bu iki boyutu için de kullanılabilir. Yani en kötü durumdaki hafıza ihtiyacı ve en kötü durumdaki hızı şeklinde algoritma analiz edilebilir. Limit teorisindeki master teoremde büyük O ile gösterilen (big-oh) değer de bu en kötü durumu analiz etmektedir. Bu yüzden en kötü durum analizine, büyük O gösterimi (Big-O notation) veya algoritmanın sonsuza giderken nasıl değiştiğini anlatmak amacıyla büyüme oranı (growth rate) isimleri verilmektedir.

REFERENCES

[1] Complexity Analysis — url: https://www.serkanseker.com/tr/algoritama-karmasiklik-analizi/
[2] url: https://www.bigocheatsheet.com/
[3] url: https://falconcoder.com/2019/11/22/algorithm-analysis-with-bigo-notation/
[4] url: https://medium.com/kodcular/nedir-bu-big-o-notation-b8b9f1416d30
[5] url: https://bilgisayarnot.blogspot.com/2020/05/algoritma-zaman-hafza-karmasiklik.html
[6] url: https://www.javatpoint.com/big-o-notation-in-c
[7] url: https://ibrahimkaya66.wordpress.com/2013/12/30/10-algoritma-analizi-algoritmalarda-karmasiklik-ve-zaman-karmasikligi/comment-page-1/