Bölüm 1: Algoritma Karmaşıklığı

Algoritmalar

Algoritma Karmaşıklığı

- *Karmaşıklık Teorisi:*
  - Bir algoritmanın kaynak kullanımını (zaman ve bellek) ölçer.
- *Büyük-O Notasyonu (Big-O Notation):*
  - Algoritmanın *en kötü durumda* çalışma süresini temsil eder.
- Örnekler:
  - $O(1)$: Sabit zamanlı
  - $O(n)$: Doğrusal zamanlı
  - $O(n^2)$: Karesel zamanlı
  - $O(\log n)$: Logaritmik zamanlı
  - $O(n \log n)$: Log-lineer zamanlı

Master Teorem

- Böl ve fethet algoritmalarının zaman karmaşıklığını çözmek için kullanılır.
- Genel Form
  - $T(n) = aT(n/b) + f(n)$
- Parametreler:
  - a: Her alt probleme bölünen kopya sayısı
  - b: Alt problemlerin boyutu
  - $f(n)$: Birleştirme süresi

Master Teorem

- Durum 1:
  - $f(n) = O(n^c)$ ve $c < \log_b a$
  - $T(n) = O(n^{\log_b a})$
- Durum 2:
  - $f(n) = O(n^c)$ ve $c = \log_b a$
  - $T(n) = O(n^{\log_b a} \log n)$
- Durum 3:
  - $f(n) = O(n^c)$ ve $c > \log_b a$
  - $T(n) = O(f(n))$

Örnek

- T(n) = 2T(n/2) + O(n)
  - $a = 2$
  - $b = 2$
  - $f(n) = O(n)$
  - $\log_b a = \log_2 2 = 1$
  - $c = 1$
- Durum 2'yi uygularız:
  - $T(n) = O(n \log n)$

f1 O(n)

```java
public int f1(int n) {
  int x = 0;
  for (int i = 0; i < n; i++) {
    x++;
  }
  return x;
}
```

f1 O(n)

- İlklendirme:
  - `int x = 0;` x değişkenine 0 atar. Sabit zamanlı işlem, O(1).
- Döngü:
  - `for (int i = 0; i < n; i++)`
  - Döngü 0’dan n’ye kadar, n kere çalışır.
- Arttırma işlemi:
  - `x++;` her döngü adımında bir kere çalışır.
  - Sabit zamanlı işlem, O(1).
- Döngü n kere çalışır, döngü gövdesi sabit zamanlı işlem yapar, toplam zaman karmaşıklığı O(n) olur.

f2 $O(n^3)$

```
public int f2(int n) {
  int x = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i * i; j++) {
      x++;
    }
  }
  return x;
}
```

f2 O(n^3)

- İlklendirme :
  - `int x = 0;` sabit zamanlı işlem, O(1).
- Dış döngü:
  - `for (int i = 0; i < n; i++)`
  - Döngü 0'dan n'ye kadar, n kere çalışır.
- İç döngü:
  - `for (int j = 0; j < i * i; j++)`
  - Değişken i, 0'dan n-1'e kadar, döngü 0'dan i * i'e kadar çalışır.
  - İterasyon sayısı i değişkeninin güncel değerine bağlıdır.

f2 $O(n^3)$

- i=0 iken: iç döngü 0 kere çalışır (0×0=0).
- i=1 iken: iç döngü 1 kere çalışır (1×1=1).
- i=2 iken: iç döngü 4 kere çalışır (2×2=4).
- i=3 iken: iç döngü 9 kere çalışır (3×3=9).

- *In general*, her i değeri için, iç döngü i^2 kere çalışır.

f2 O(n^3)

- İç döngünün toplam iterasyon sayısı (0’dan n−1’e kadar):
    - $\sum_0^{n-1} i^2$
    - $\frac{(n-1)n(2n-1)}{6}$
    - *simplifies to* n^3 / 3.
- *Therefore*, zaman karmaşıklığı: O(n^3)

f3 $O(2^n)$

```java
public int f3(int n) {
  if (n <= 1) {
    return 1;
  }
  return f3(n - 1) + f3(n - 1);
}
```

f3 $O(2^n)$

- Temel durum (*base case*):
  - $n \leq 1$ iken, fonksiyon 1 döner. Sabit zamanlı işlem, $O(1)$.
- Özyinelemeli durum (*recursive case*):
  - $n > 1$ iken, fonksiyon iki özyinelemeli çağrı yapar $f3(n-1)$.
  - *This creates a recurrence relation*:
    - $T(n) = 2T(n-1)$
  - Temel durum:
    - $T(n) = O(1)$, $n \leq 1$ için.

f3 $O(2^n)$

- $T(n) = 2T(n-1) = 2\cdot2T(n-2) = 2\cdot2\cdot2T(n-3) = 2^k\, T(n-k)$
- Denklemi $n-k=0$ oluncaya kadar açarsak:
  - $T(n) = 2^n\, T(0)$,
  - $T(0) = 1$ olduğundan,
  - $T(n) = 2^n$ olur.

f4 O(n)

```java
public int f4(int n) {
  if (n <= 1) {
    return 1;
  }
  return f4(n / 2) + f4(n / 2);
}
```

f4 O(n)

- Temel durum (*base case*):
  - $n \le 1$ iken, fonksiyon 1 döner. Sabit zamanlı işlem, $O(1)$.
- Özyinelemeli durum (*recursive case*):
  - $n > 1$ iken, fonksiyon özyinelemeli iki çağrı yapar f4(n/2).
  - *This creates a recurrence relation*:
    - $T(n) = 2T(n/2)$
  - Temel durum:
    - $T(n) = O(1)$, $n \le 1$ için.

f4 O(n)

- $T(n) = a\,T(n/b) + f(n)$
- Örnekte, $a=2$, $b=2$, $f(n)=O(1)$, $\log_b a = \log_2 2 = 1$.
- *Here*, $f(n) = O(1)$ *corresponds to* $c = 0$, *which is less than* $\log_b a = 1$.
- According to the Master Theorem,
  - if $f(n) = O(n^c)$ *where* $c < \log_b a$, *then* $T(n) = O(n^{\log_b a})$.
- *Therefore*:
  - $T(n) = O(n^{\log_2 2}) = O(n^1) = O(n)$

f5 O(nlogn)

```
public int f5(int n) {
  if (n <= 1) {
    return 1;
  }
  return f1(n) + f5(n / 2) + f5(n / 2);
}
```

f5 O(nlogn)

- Temel durum:
  - n≤1 iken, fonksiyon 1 döner. Sabit zamanlı işlem, O(1).
- Özyinelemeli durum:
  - n>1 iken, fonksiyon f1(n)’i çağırır, ve iki çağrı daha yapar f5(n/2).

- $T(n) = 2T(n/2) + f1(n)$
- $T(n) = 2T(n/2) + O(n)$ ($\underline{\textit{T(n) = aT(n/b) + f(n)}}$)
- $a = 2$, $b = 2$, $f(n) = O(n)$, $\log_b a = \log_2 2 = 1$
- $f(n) = O(n)$ *corresponds to* $c=1$
- $T(n) = O(n^{\log_b a} \log n) = O(n\log n)$

f6 O(logn)

```java
public static int f6(int n) {
  int x = 0;
  // 1<<i is the same as 2^i
  // Ignore integer overflow.
  // 1<<i takes constant time.
  for (int i = 0; i < n; i = 1 << i) {
    x++;
  }
  return x;
}
```

f6 O(logn)

- İlklendirme (*initialization*):
  - `int i = 0;` i değişkenine 0 atar. Sabit zamanlı işlem, O(1).
- Koşul (*condition*):
  - `i < n` ifadesi, her iterasyonda i'nin n'den küçük olduğunu kontrol eder.
- Güncelleme (*update*):
  - `i = 1 << i` ifadesi i değerini 2^i olarak günceller.
  - *(since 1 << i is the same as* 2^i*).*

f6 O(logn)

- Başlangıçta, i = 0.
- İlk adım sonrası, $i = 2^0 = 1$.
- İkinci adım sonrası, $i = 2^1 = 2$.
- Üçüncü adım sonrası, $i = 2^2 = 4$.
- Dördüncü adım sonrası, $i = 2^4 = 16$.

- Değişken i değeri hızlı büyür. (*exponential nature of* 2^i).

f6 O(logn)

- n değerine erişmek için gereken adım sayısı k olsun. $2^k \geq n$
- İki tarafın logaritması alınırsa:
    - $k \geq \log_2 n$
- *Therefore*, k değeri $\log_2 n$ değerine yaklaşır.
- *Hence*, f6 fonksiyonunun zaman karmaşıklığı O(logn) olur.

SON