# Bölüm 1: Algoritma Karmaşıklığı

## Algoritmalar

# Algoritma Karmaşıklığı

- *Karmaşıklık Teorisi:*
  - Bir algoritmanın kaynak kullanımını (zaman ve bellek) ölçer.
- *Büyük-O Notasyonu (Big-O Notation):*
  - En kötü durumda bir algoritmanın 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 Teoremi

- 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 Teoremi

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

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

# f1 O(n)

- Initialization:
  - int x = 0; initializes the variable x to 0. constant time operation, O(1).
- Loop:
  - for (int i = 0; i < n; i++)
  - The loop runs from 0 to n, so it iterates n times.
- Increment Operation:
  - x++; is executed once per iteration of the loop.
  - This is a constant time operation, O(1).
- Since the loop runs n times and the body of the loop performs a constant time operation, the total time complexity is O(n).

# 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 $O(n^3)$

- Initialization:
  - int x = 0; is a constant time operation, O(1).
- Outer Loop:
  - for (int i = 0; i < n; i++)
  - This loop runs from 0 to n, so it iterates n times.
- Inner Loop:
  - for (int j = 0; j < i * i; j++)
  - For each value of i from 0 to n-1, the inner loop runs from 0 to i * i.
  - Therefore, the number of iterations depends on the current value of i.

# f2 $O(n^3)$

- When i=0: the inner loop runs 0 times (since 0×0=0).
- When i=1: the inner loop runs 1 times (since 1×1=1).
- When i=2: the inner loop runs 4 times (since 2×2=4).
- When i=3: the inner loop runs 9 times (since 3×3=9).

- In general, for each i, the inner loop runs $i^2$ times.

# f2 $O(n^3)$

- The total number of iterations of the inner loop from 0 to n−1:
    - $\sum_{0}^{n-1} i^2$
    - $\frac{(n-1)n(2n-1)}{6}$
    - simplifies to $n^3 / 3$.
- Therefore, the time complexity is: $O(n^3)$

# f3 $O(2^n)$

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

# f3 $O(2^n)$

- Base Case:
  - When $n \leq 1$, the function returns 1. constant time operation, $O(1)$.
- Recursive Case:
  - When $n > 1$, the function makes two recursive calls $f3(n-1)$.
  - This creates a recurrence relation:
    - $T(n) = 2T(n-1)$
  - The base case is:
    - $T(n) = O(1)$ for $n \leq 1$

# f3 $O(2^n)$

- $T(n) = 2T(n-1) = 2 \cdot 2T(n-2) = 2 \cdot 2 \cdot 2T(n-3) = 2^k\ T(n-k)$
- continue expanding until $n-k=0$:
  - $T(n) = 2^n\ T(0)$
  - Since $T(0) = 1$
  - $T(n) = 2^n$

# f4 O(n)

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

# f4 O(n)

- Base Case:
  - When $n \leq 1$, the function returns 1. constant time operation, O(1).
- Recursive Case:
  - When $n > 1$, the function makes two recursive calls f4(n/2).
  - This creates a recurrence relation:
    - $T(n) = 2T(n/2)$
  - The base case is:
    - $T(n) = O(1)$ for $n \leq 1$

# f4 O(n)

- T(n) = a T(n/b) + f(n)
- In our case, 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)

- Base Case:
  - When $n \leq 1$, the function returns 1. constant time operation, O(1).
- Recursive Case:
  - When $n > 1$, function calls f1(n) and makes two recursive calls f5(n/2).

- $T(n) = 2T(n/2) + f1(n)$
- $T(n) = 2T(n/2) + O(n)$ ($\underline{T(n) = aT(n/b) + f(n)}$)
- $a = 2$, $b = 2$, $f(n) = O(n)$, $\log_b a = \log_2 2 = 1$
- $f(n) = O(n)$ 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)

- Initialization:
  - int i = 0; initializes i to 0. constant time operation, O(1).
- Condition:
  - i < n checks if i is less than n, at each iteration.
- Update:
  - i = 1 << i updates i to $2^i$ (since 1 << i is the same as $2^i$).

# f6 O(logn)

- Initially, $i = 0$.
- After the first iteration, $i = 2^0 = 1$.
- After the second iteration, $i = 2^1 = 2$.
- After the third iteration, $i = 2^2 = 4$.
- After the fourth iteration, $i = 2^4 =16$.

- The value of i grows extremely quickly due to the exponential nature of $2^i$.

# f6 O(logn)

- Let k be the number of iterations needed to reach or exceed n. $2^k \geq n$
- Taking the logarithm of both sides:
  - $k \geq \log_2 n$
- Therefore, the number of iterations k is approximately $\log_2 n$.
- Hence, the time complexity of the function f6 is O(logn).

SON