

BİL3014

Algoritma Analizi

Dr. Öğr. Üyesi Emre DELİBAŞ

Dinamik Programlama



“

Dinamik Programlama

Bilgisayar bilimi, matematik, ekonomi ve biyoinformatikte dinamik programlama (ya da dinamik optimizasyon) karmaşık bir problemi tekrarlanan alt problemlere bölerek, her bir alt problemi yalnız bir kere çözüp daha sonra bu çözümü kaydederek karmaşık problemin çözümünde kullanma yöntemidir.



”

- Bu teknik, problemin özelliklerini kullanarak alt sorunların çözümlerini hesaplamak için daha önceden hesaplanmış sonuçları kullanır.
- Bu sayede alt sorunların tekrar tekrar çözülmesi önlenir ve problemin çözümü daha hızlı bir şekilde elde edilebilir.

Dinamik Programlama

- Dinamik Programlama, genellikle "optimal substructure" ve "overlapping subproblems" özelliklerini sağlayan problemlerin çözümünde kullanılır.
- ***Optimal substructure*** özelliği, bir problemin optimal çözümünün, alt problemlerin optimal çözümlerinin birleştirilmesi ile elde edilebileceğini ifade eder.
- ***Overlapping subproblems*** özelliği ise, birden fazla alt problemin aynı alt problemleri içermesi durumudur.



Böl-Yönet ve Dinamik Programlama

- Dinamik programlama (DP) ve Böl-Yönet (Divide and Conquer) algoritmaları, karmaşık problemleri daha küçük alt problemlere bölerek çözmeye çalışırlar.
- Ancak bu iki yaklaşım arasında bazı farklılıklar vardır:
 - Dinamik programlama, alt problemleri çözerek bir üst seviye problemi çözmeye çalışır. Yani alt problemlerin çözümleri daha sonra tekrar kullanılmak üzere bir tabloda saklanır ve daha büyük problemleri çözmek için kullanılır.
 - Böl-Yönet yaklaşımı ise, bir problemin alt problemlere bölünmesini ve her alt problemi ayrı ayrı çözülmesini sağlar.
 - Böl-Yönet yaklaşımında alt çözümler birleştirilir, dinamik programlamada alt çözümler kullanılır.



Böl-Yönet ve Dinamik Programlama

- Dinamik programlama ve Böl-Yönet yaklaşımları, her birinin veri yapısı ve probleme bağlı olarak farklı avantajları ve dezavantajları vardır.
- Böl-Yönet yaklaşımı, alt problemler birbirinden bağımsız olduğu için paralelleştirilebilir, ancak bazı durumlarda alt problemlerin birbirine bağımlı olması nedeniyle kullanılamaz.
- Dinamik programlama, alt problemler arasındaki bağımlılıkların hesaplanması için bir tablo kullanır, bu nedenle bazı durumlarda daha fazla hafıza kullanabilir ve daha yavaş olabilir. Ancak bazı problemlerde DP çözümü daha verimli kullanılabilir ve hatta tek seçenek olabilir.



SRTBOT

- Dinamik programlama çözümleriyle uyumlu rekursif yaklaşımların çözüm stratejisini hatırlayalım:
 - Sub-problems definition
 - Relate subproblem solutions recursively
 - Topological Order on subproblems
 - Base cases or relation
 - Original Problem solution via subproblem(s)
 - Time analysis
- Dinamik programlama daha önce yaptığınız işi yeniden kullanma mantığı olan ***memoization*** adlı yeni bir fikir ekleyerek bu şablon üzerine inşa edilecektir.



Top-down ve Bottom-up Yaklaşımları

- Dinamik programlama yaklaşımı iki farklı şekilde uygulanabilir: top-down ve bottom-up.
-
- **Top-down (recursion + memoization)** yaklaşımında, bir problem bir ana sorun olarak ele alınır ve daha küçük problemlere bölünür.
 - Alt problemler çözüldükten sonra, çözümler kaydedilir ve tekrar kullanılır. (remember & re-use)
 - Bu yöntemde genellikle özyinelemeli bir yapı kullanılır.
 - **Bottom-up (tabulation)** yaklaşımında, alt problemler önceden çözülür ve daha sonra ana problemi çözmek için bu sonuçlar kullanılır.
 - Alt problemler topolojik sıralama düzeninde çözülür. (genellikle döngülerle)

Fibonacci Sayıları

- Fibonacci Sayıları 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

- Amaç: n verildiğinde F_n i hesapla

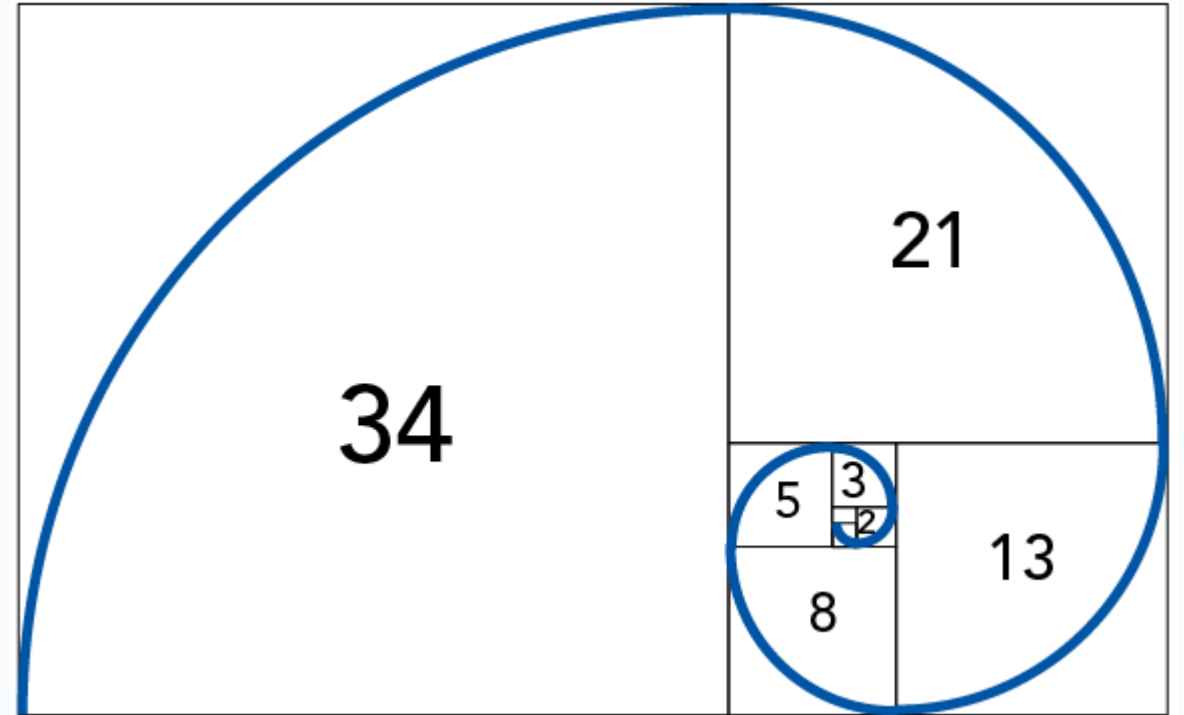
$$F_0 = 0 \text{ ve } F_1 = 1$$

Input : $n = 2$

Output : 1

Input : $n = 9$

Output : 34



Fibonacci Sayıları

$$F_n = F_{n-1} + F_{n-2} \qquad F_1 = F_2 = 1$$

- Subproblems : $F(i) = i$. Fibonacci sayısı F_i , $i \in \{0, 1, \dots, n\}$
- Relation : $F(i) = F(i - 1) + F(i - 2)$ (Fibonacci say. tanımı)
- Topo. Order : i yi artır , *for* $i=1, 2, \dots, n$
- Base cases : $F(0) = 0, F(1) = 1$
- Original prob. : $F(n)$

```
1 def fib(n):  
2     if n < 2: return n                # base case  
3     return fib(n - 1) + fib(n - 2)    # recurrence
```

- Time: $T(n) = T(n - 1) + T(n - 2) + O(1) > 2T(n - 2)$, $T(n) = \Omega(2^{n/2})$ exp. :(
- Alt problem $F(k)$ bir kereden fazla hesaplandı! ($F(n - k)$ kez)

Fibonacci Sayıları

- Fibonacci Sayıları 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

- Amaç: n verildiğinde F_n i hesapla

- Recursive yaklaşım:

```
public static long fib(int i)
{
    if (i == 0) return 0;
    if (i == 1) return 1;
    return fib(i-1) + fib(i-2);
}
```

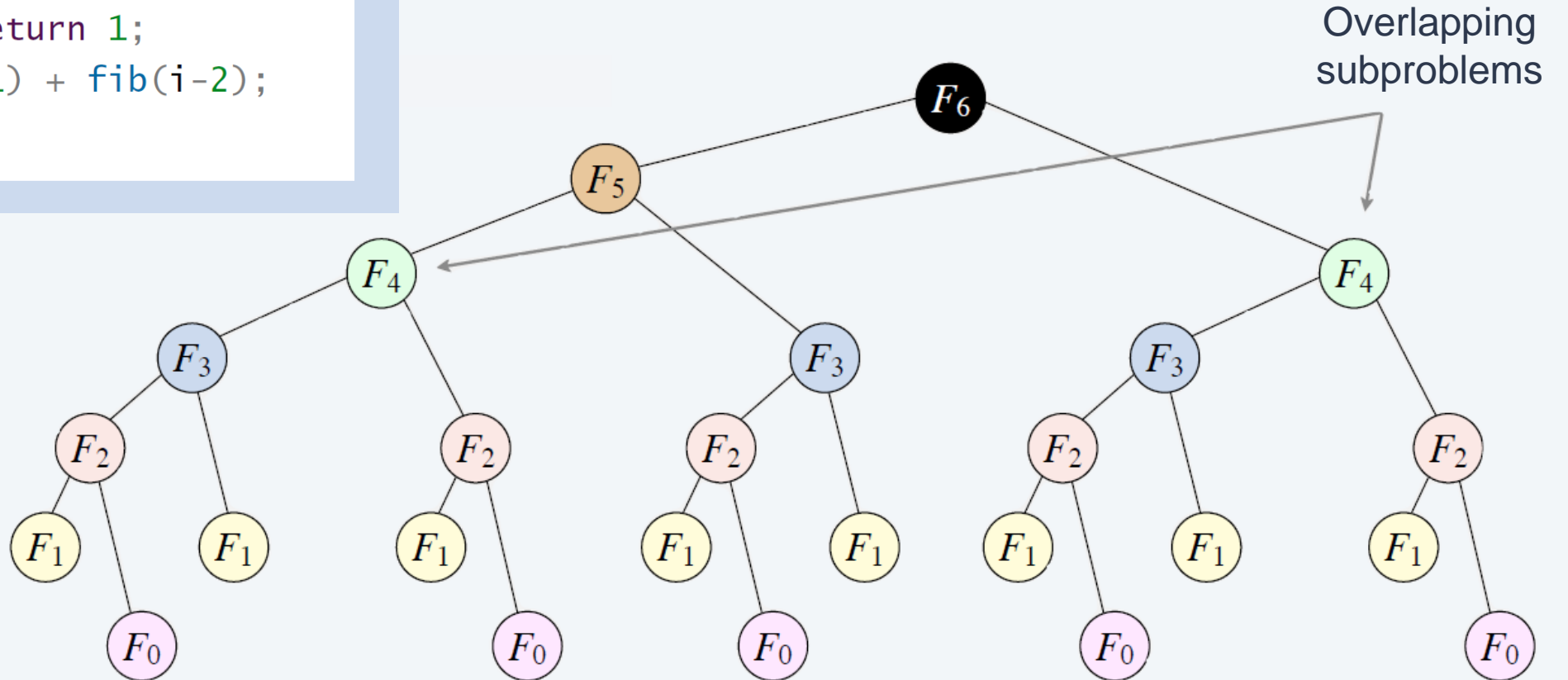
$$F_0 = 0 \text{ ve } F_1 = 1$$

Fibonacci Sayıları

Recursion kullanımı

```
public static long fib(int i)
{
    if (i == 0) return 0;
    if (i == 1) return 1;
    return fib(i-1) + fib(i-2);
}
```

$$\theta(n) = 2^n$$



Fibonacci Sayıları

Örtüşen problemler tekrar tekrar çözülür.

Karmaşıklık : 2^n

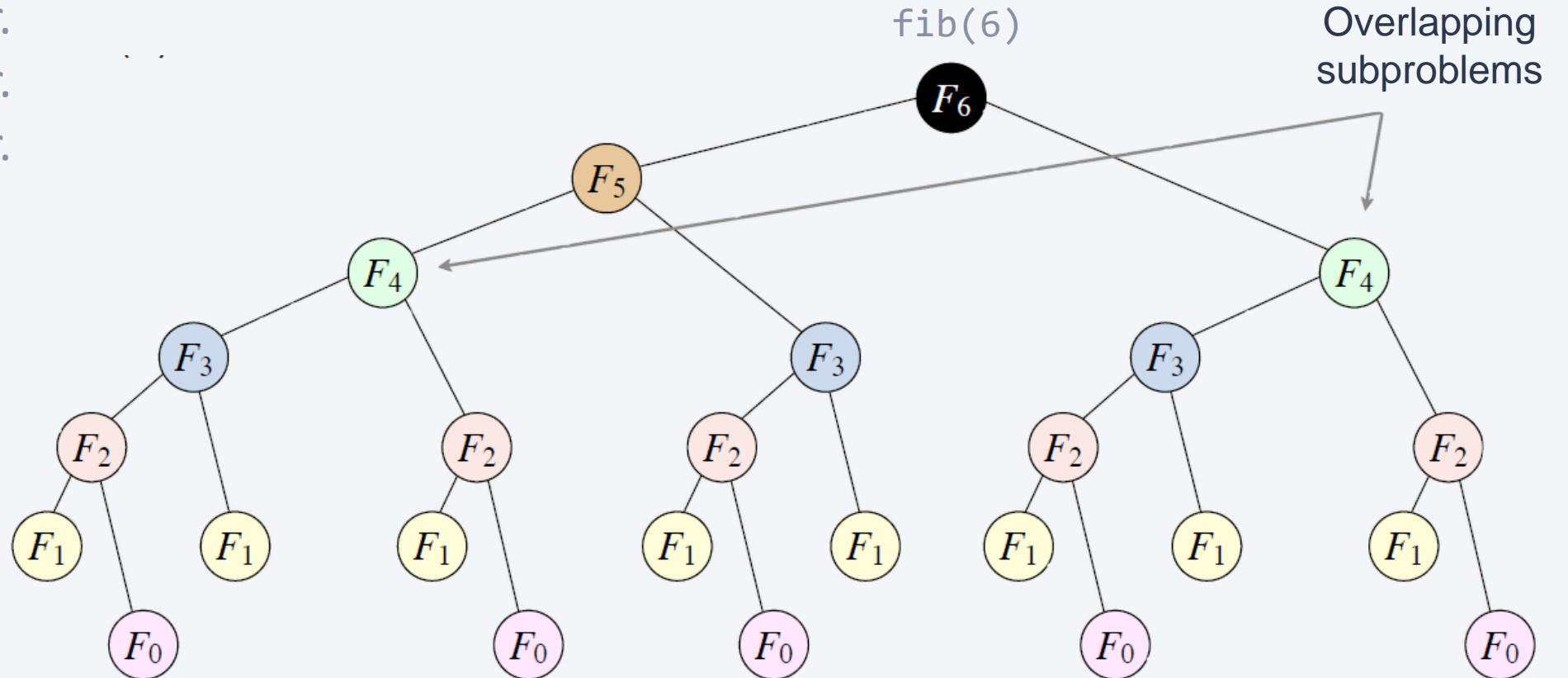
fib(5) 1 kez çağırılır.

fib(4) 2 kez çağırılır.

fib(3) 3 kez çağırılır.

fib(2) 5 kez çağırılır.

fib(1) 8 kez çağırılır.



Fibonacci Sayıları

Memoization

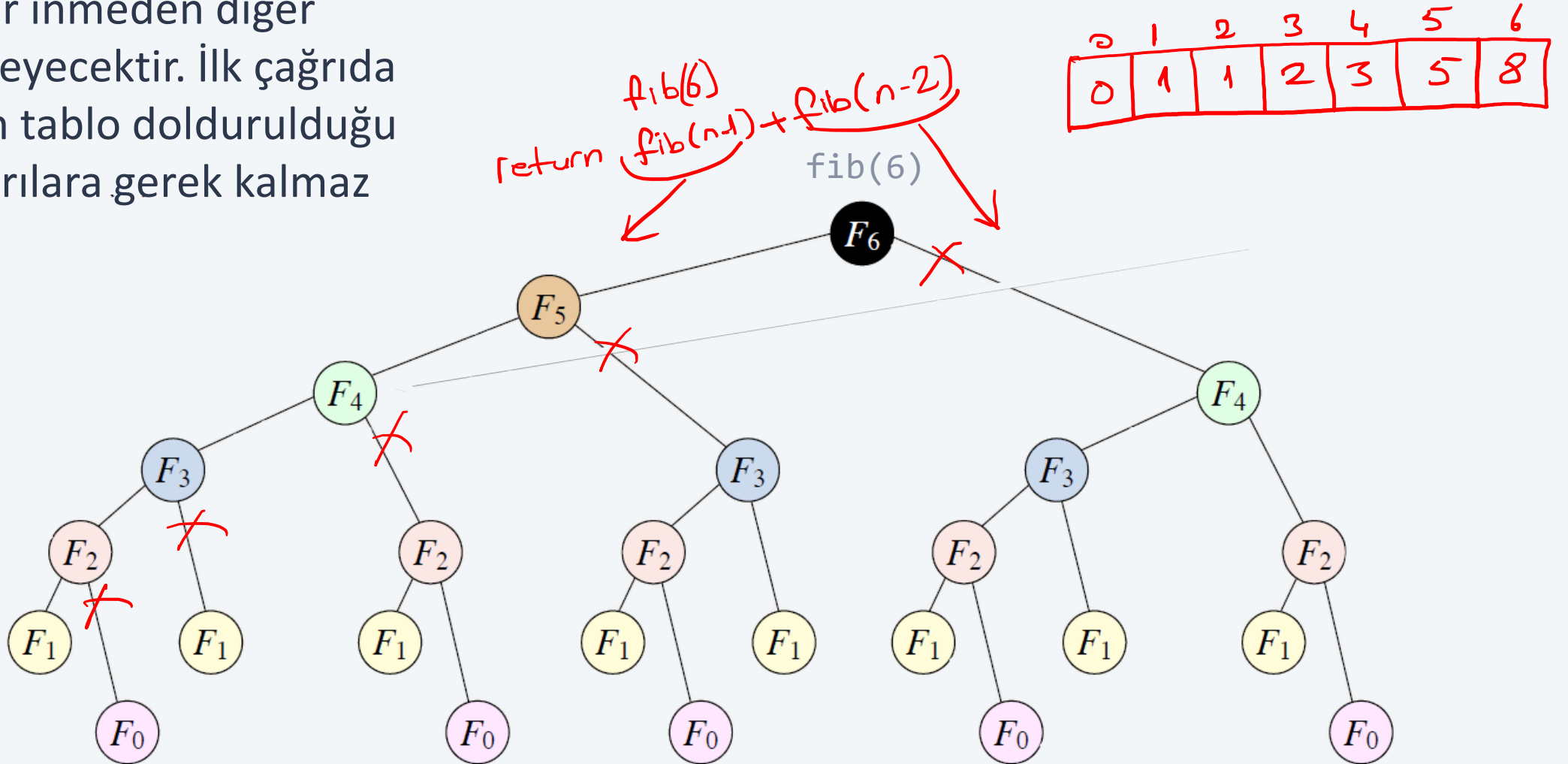
- Hesaplanan tüm değerleri hatırlama için bir dizi (veya bir sembol tablosu) tutun.
- Hesaplanacak değer biliniyorsa, onu döndürmeniz yeterli; aksi halde hesaplayın, not alın ve döndürün.

```
public static long fib(int i)
{
    if (i == 0) return 0;
    if (i == 1) return 1;
    if (f[i] == 0) f[i] = fib(i-1) + fib(i-2);
    return f[i];
}
```

$$\theta(n) = n$$

Fibonacci Sayıları

Alt çağrılardan önce çağrılan base duruma kadar inmeden diğer çağrıya geçmeyecektir. İlk çağrıda ($\text{fib}(n-1)$) tüm tablo doldurulduğu için ikinci çağrılara gerek kalmaz

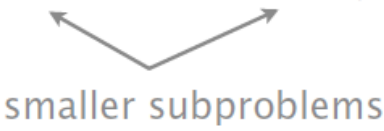


Fibonacci Sayıları

Bottom-up Dinamik Programlama

- Hesaplamayı "aşağıdan yukarıya" oluşturun.
- Küçük alt problemleri çözün ve çözümleri kaydedin.
- Daha büyük alt problemleri çözmek için bu çözümleri kullanın.

```
public static long fib(int n)
{
    long[] f = new long[n+1];
    f[0] = 0;
    f[1] = 1;
    for (int i = 2; i <= n; i++)
        f[i] = f[i-1] + f[i-2];
    return f[n];
}
```



$$\theta(n) = n$$

Fibonacci Sayıları

Performans Artırmak için;

- Yalnızca en yeni iki Fibonacci sayısını kaydederek yerden tasarruf edin.

```
public static long fib(int n) {  
    int f = 1, g = 0;  
    for (int i = 1; i < n-1; i++) {  
        f = f + g;  
        g = f - g;  
    }  
    return f;  
}
```

← F ve g ardışık fibonacci sayılarıdır.