

# Divide and Conquer

# Divide and Conquer

- **Divide:** Decompose an instance of a problem into smaller instances of the same problem
- **Conquer:** Solve each subproblem (recursively)
- **Recombine** the subproblems' solutions into a solution to the original problem



# Polynomial Multiplication

**Problem:** Given two polynomials  $P(x), Q(x)$  of degree  $n$ , compute  $R(x) = P(x) \cdot Q(x)$

**Instance:**

- The coefficients  $p_0, p_1, \dots, p_n \in \mathbb{Z}$  of  $P(x) = \sum_{i=0}^n p_i x^i$ .
- The coefficients  $q_0, q_1, \dots, q_n \in \mathbb{Z}$  of  $Q(x) = \sum_{i=0}^n q_i x^i$ .

**Solution:**

- The coefficients  $r_0, r_1, \dots, r_{2n} \in \mathbb{Z}$  of

$$R(x) = P(x) \cdot Q(x) = \sum_{i=0}^{2n} r_i x^i.$$

(Assume that arithmetic operations can be performed in  $O(1)$  time).

# Example

$$P(x) = 1 + 2x + 3x^2$$

$$Q(x) = 3 + 0x + 5x^2$$

$$R(x) = P(x) \cdot Q(x) = 3 + 6x + 14x^2 + 10x^3 + 15x^4$$

How to compute  $R(x)$  efficiently?

# Intermission: A More General Problem

Given two binary operations  $\oplus, \otimes$  and two functions  $f, g : \mathbb{Z} \rightarrow \mathbb{R}$ , the  $(\oplus, \otimes)$ -discrete convolution of  $f$  and  $g$  is a function  $(f * g) : \mathbb{Z} \rightarrow \mathbb{R}$  defined as:

$$(f * g)(n) = \bigoplus_{m=-\infty}^{+\infty} \left( f(n - m) \otimes g(m) \right)$$

# Intermission: A More General Problem

Given two binary operations  $\oplus, \otimes$  and two functions  $f, g : \mathbb{Z} \rightarrow \mathbb{R}$ , the  $(\oplus, \otimes)$ -discrete convolution of  $f$  and  $g$  is a function  $(f * g) : \mathbb{Z} \rightarrow \mathbb{R}$  defined as:

$$(f * g)(n) = \bigoplus_{m=-\infty}^{+\infty} \left( f(n - m) \otimes g(m) \right)$$

Consider the arrays  $P$  and  $Q$  associated with the polynomials  $P(x)$  and  $Q(x)$ . Define  $f(n) = p_n$ ,  $g(n) = q_n$  (and 0 elsewhere). The  $(+, \cdot)$  convolution of  $P$  and  $Q$  is:

$$r_n = (f * g)(n) = \sum_{m=0}^n p_{n-m} q_m$$

# Back to Polynomials: A Trivial Solution

$$r_i = \sum_{j=0}^i p_{i-j} q_j$$

- For  $i = 0, \dots, 2n$  :
  - $r_i \leftarrow 0$
  - For  $j = \max\{0, i - n\}, \dots, \min\{i, n\}$  :
    - $r_i \leftarrow r_i + p_{i-j} \cdot q_j$

# Back to Polynomials: A Trivial Solution

$$r_i = \sum_{j=0}^i p_{i-j} q_j$$

- For  $i = 0, \dots, 2n$  :
  - $r_i \leftarrow 0$
  - For  $j = \max\{0, i - n\}, \dots, \min\{i, n\}$  :
    - $r_i \leftarrow r_i + p_{i-j} \cdot q_j$

Time Complexity:  $\Theta(n^2)$

# Back to Polynomials: A Trivial Solution

$$r_i = \sum_{j=0}^i p_{i-j} q_j$$

- For  $i = 0, \dots, 2n$  :
  - $r_i \leftarrow 0$
  - For  $j = \max\{0, i - n\}, \dots, \min\{i, n\}$  :
    - $r_i \leftarrow r_i + p_{i-j} \cdot q_j$

Time Complexity:  $\Theta(n^2)$

Can we do better?

# Divide and Conquer: First Attempt

- Write  $P$  as:  $P(x) = P'(x) + P''(x) \cdot x^{\lfloor n/2 \rfloor}$  , where:

$$P'(x) = \sum_{i=0}^{\lfloor n/2 \rfloor} p_i x^i \quad \text{and} \quad P''(x) = \sum_{i=1+\lfloor n/2 \rfloor}^n p_i x^{i-\lfloor n/2 \rfloor}$$

# Divide and Conquer: First Attempt

- Write  $P$  as:  $P(x) = P'(x) + P''(x) \cdot x^{\lfloor n/2 \rfloor}$ , where:

$$P'(x) = \sum_{i=0}^{\lfloor n/2 \rfloor} p_i x^i \quad \text{and} \quad P''(x) = \sum_{i=1+\lfloor n/2 \rfloor}^n p_i x^{i-\lfloor n/2 \rfloor}$$

- Similarly, write  $Q$  as:  $Q(x) = Q'(x) + Q''(x) \cdot x^{\lfloor n/2 \rfloor}$

# Divide and Conquer: First Attempt

- Write  $P$  as:  $P(x) = P'(x) + P''(x) \cdot x^{\lfloor n/2 \rfloor}$ , where:

$$P'(x) = \sum_{i=0}^{\lfloor n/2 \rfloor} p_i x^i \quad \text{and} \quad P''(x) = \sum_{i=1+\lfloor n/2 \rfloor}^n p_i x^{i-\lfloor n/2 \rfloor}$$

- Similarly, write  $Q$  as:  $Q(x) = Q'(x) + Q''(x) \cdot x^{\lfloor n/2 \rfloor}$

$$P(x) \cdot Q(x) = (P'(x) + P''(x) \cdot x^{\lfloor n/2 \rfloor}) \cdot (Q'(x) + Q''(x) \cdot x^{\lfloor n/2 \rfloor})$$

# Divide and Conquer: First Attempt

- Write  $P$  as:  $P(x) = P'(x) + P''(x) \cdot x^{\lfloor n/2 \rfloor}$ , where:

$$P'(x) = \sum_{i=0}^{\lfloor n/2 \rfloor} p_i x^i \quad \text{and} \quad P''(x) = \sum_{i=1+\lfloor n/2 \rfloor}^n p_i x^{i-\lfloor n/2 \rfloor}$$

- Similarly, write  $Q$  as:  $Q(x) = Q'(x) + Q''(x) \cdot x^{\lfloor n/2 \rfloor}$

$$\begin{aligned} P(x) \cdot Q(x) &= (P'(x) + P''(x) \cdot x^{\lfloor n/2 \rfloor}) \cdot (Q'(x) + Q''(x) \cdot x^{\lfloor n/2 \rfloor}) \\ &= P'(x)Q'(x) + (P'(x)Q''(x) + P''(x)Q'(x))x^{\lfloor n/2 \rfloor} + P''(x)Q''(x)x^{2\lfloor n/2 \rfloor} \end{aligned}$$

# Divide and Conquer: First Attempt

$$P'(x)Q'(x) + (P'(x)Q''(x) + P''(x)Q'(x))x^{\lfloor n/2 \rfloor} + P''(X)Q''(x)x^{2\lfloor n/2 \rfloor}$$

The problem of computing the product of two polynomials of degree  $n$  is reduced to that of computing 4 products of polynomials of degree  $\approx n/2$ .

# Divide and Conquer: First Attempt

$$P'(x)Q'(x) + (P'(x)Q''(x) + P''(x)Q'(x))x^{\lfloor n/2 \rfloor} + P''(x)Q''(x)x^{2\lfloor n/2 \rfloor}$$

The problem of computing the product of two polynomials of degree  $n$  is reduced to that of computing 4 products of polynomials of degree  $\approx n/2$ .

## Recurrence Equation:

$$T(n) = 4T(n/2) + O(n)$$


$O(n)$  time is needed to decompose the polynomials and to recombine the 4 sub-products.

# Divide and Conquer: First Attempt

$$P'(x)Q'(x) + (P'(x)Q''(x) + P''(x)Q'(x))x^{\lfloor n/2 \rfloor} + P''(x)Q''(x)x^{2\lfloor n/2 \rfloor}$$

The problem of computing the product of two polynomials of degree  $n$  is reduced to that of computing 4 products of polynomials of degree  $\approx n/2$ .

## Recurrence Equation:

$$T(n) = 4T(n/2) + O(n)$$


$O(n)$  time is needed to decompose the polynomials and to recombine the 4 sub-products.

**Solution:**  $\Theta(n^2)$

# Divide and Conquer: First Attempt

$$P'(x)Q'(x) + (P'(x)Q''(x) + P''(x)Q'(x))x^{\lfloor n/2 \rfloor} + P''(x)Q''(x)x^{2\lfloor n/2 \rfloor}$$

The problem of computing the product of two polynomials of degree  $n$  is reduced to that of computing 4 products of polynomials of degree  $\approx n/2$ .



Still  $\Theta(n^2)$

**Solution:**  $\Theta(n^2)$

# Divide and Conquer: Second Attempt

**We want:**

$$P'(x)Q'(x) + (P'(x)Q''(x) + P''(x)Q'(x))x^{\lfloor n/2 \rfloor} + P''(X)Q''(x)x^{2\lfloor n/2 \rfloor}$$

# Divide and Conquer: Second Attempt

**We want:**

$$P'(x)Q'(x) + (P'(x)Q''(x) + P''(x)Q'(x))x^{\lfloor n/2 \rfloor} + P''(x)Q''(x)x^{2\lfloor n/2 \rfloor}$$

**Define:**

$$U = P'(x)Q'(x) \quad V = P''(x)Q''(x)$$

$$W = (P'(x) + P''(x))(Q'(x) + Q''(x))$$

# Divide and Conquer: Second Attempt

**We want:**

$$P'(x)Q'(x) + \underbrace{(P'(x)Q''(x) + P''(x)Q'(x))}_{\downarrow} x^{\lfloor n/2 \rfloor} + \underbrace{P''(x)Q''(x)}_{\downarrow} x^{2\lfloor n/2 \rfloor}$$

$U$    $V$

**Define:**

$$U = P'(x)Q'(x) \quad V = P''(x)Q''(x)$$

$$W = (P'(x) + P''(x))(Q'(x) + Q''(x))$$





# Divide and Conquer: Second Attempt

- **Divide:**

$$U = P'(x) \cdot Q'(x) \quad (\text{subproblem 1})$$

$$V = P''(x) \cdot Q''(x) \quad (\text{subproblem 2})$$

$$W = (P'(x) + P''(x)) \cdot (Q'(x) + Q''(x)) \quad (\text{subproblem 3})$$

- **Conquer:** Compute  $U, V, W$  recursively

- **Recombine:**  $U + (W - U - V)x^{\lfloor n/2 \rfloor} + Vx^{2\lfloor n/2 \rfloor}$



# Divide and Conquer: Second Attempt

- **Divide:**

$$U = P'(x) \cdot Q'(x) \quad (\text{subproblem 1})$$

$$V = P''(x) \cdot Q''(x) \quad (\text{subproblem 2})$$

$$W = (P'(x) + P''(x)) \cdot (Q'(x) + Q''(x)) \quad (\text{subproblem 3})$$

- **Conquer:** Compute  $U, V, W$  recursively

- **Recombine:**  $U + (W - U - V)x^{\lfloor n/2 \rfloor} + Vx^{2\lfloor n/2 \rfloor}$

**Reurrence Equation:**  $T(n) = 3T(n/2) + O(n)$



# Divide and Conquer: Second Attempt

- **Divide:**

$$U = P'(x) \cdot Q'(x) \quad (\text{subproblem 1})$$

$$V = P''(x) \cdot Q''(x) \quad (\text{subproblem 2})$$

$$W = (P'(x) + P''(x)) \cdot (Q'(x) + Q''(x)) \quad (\text{subproblem 3})$$

- **Conquer:** Compute  $U, V, W$  recursively

- **Recombine:**  $U + (W - U - V)x^{\lfloor n/2 \rfloor} + Vx^{2\lfloor n/2 \rfloor}$

**Reurrence Equation:**  $T(n) = 3T(n/2) + O(n)$

**Solution:**  $O(n^{\log_2 3}) = O(n^{1.585})$



# Divide and Conquer: Second Attempt

- **Divide:**

$$U = P'(x) \cdot Q'(x) \quad (\text{subproblem 1})$$

$$V = P''(x) \cdot Q''(x) \quad (\text{subproblem 2})$$

$$W = (P'(x) + P''(x)) \cdot (Q'(x) + Q''(x)) \quad (\text{subproblem 3})$$

- **Conquer:** Compute  $U, V, W$  recursively

- **Recombine:**  $U + (W - U - V)x^{\lfloor n/2 \rfloor} + Vx^{2\lfloor n/2 \rfloor}$

## Trick/Technique: Divide and Conquer

Decompose an instance into smaller instances of the same problem.

Solve recursively and recombine the solutions.

# Recursion & Memoization

# Fibonacci Numbers

**Definition:**  $F_0 = 0$ ,  $F_1 = 1$ , and  $F_n = F_{n-1} + F_{n-2}$  for  $n > 1$

**Problem:** Given  $n \in \mathbb{N}$ , compute  $F_n$

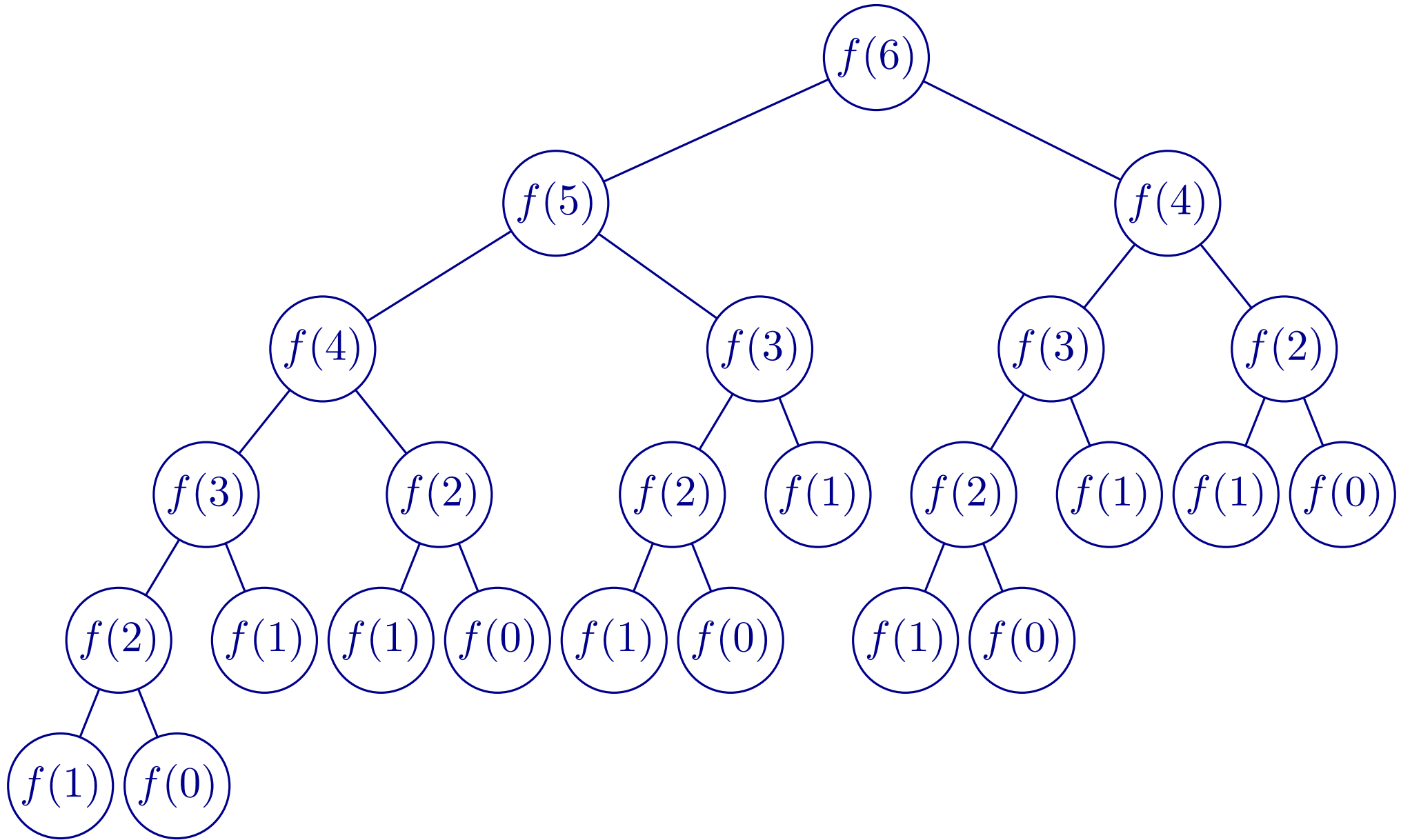
A trivial **recursive** solution:

```
int fibonacci(int n)
{
    if (n <= 1)
        return n;

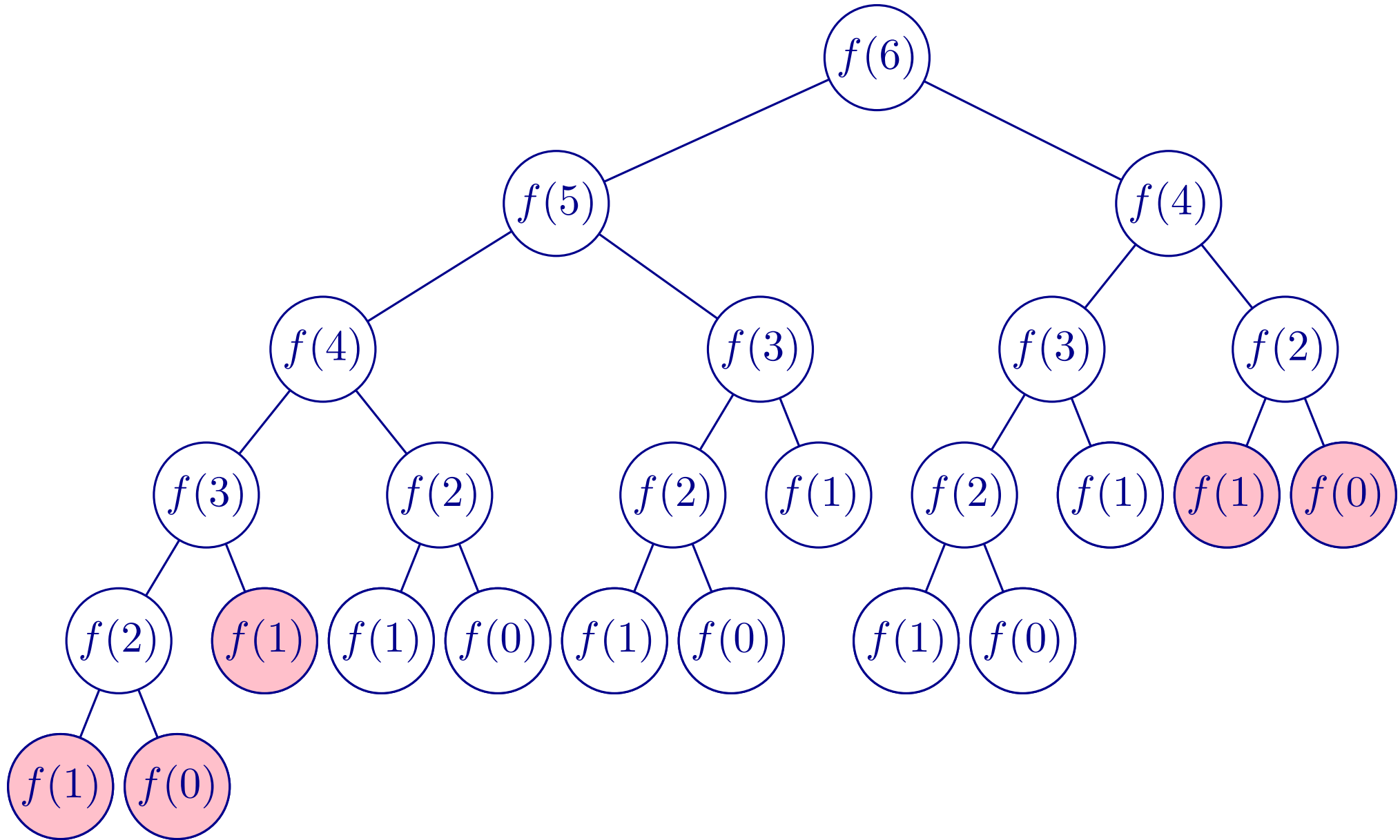
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Computational complexity?

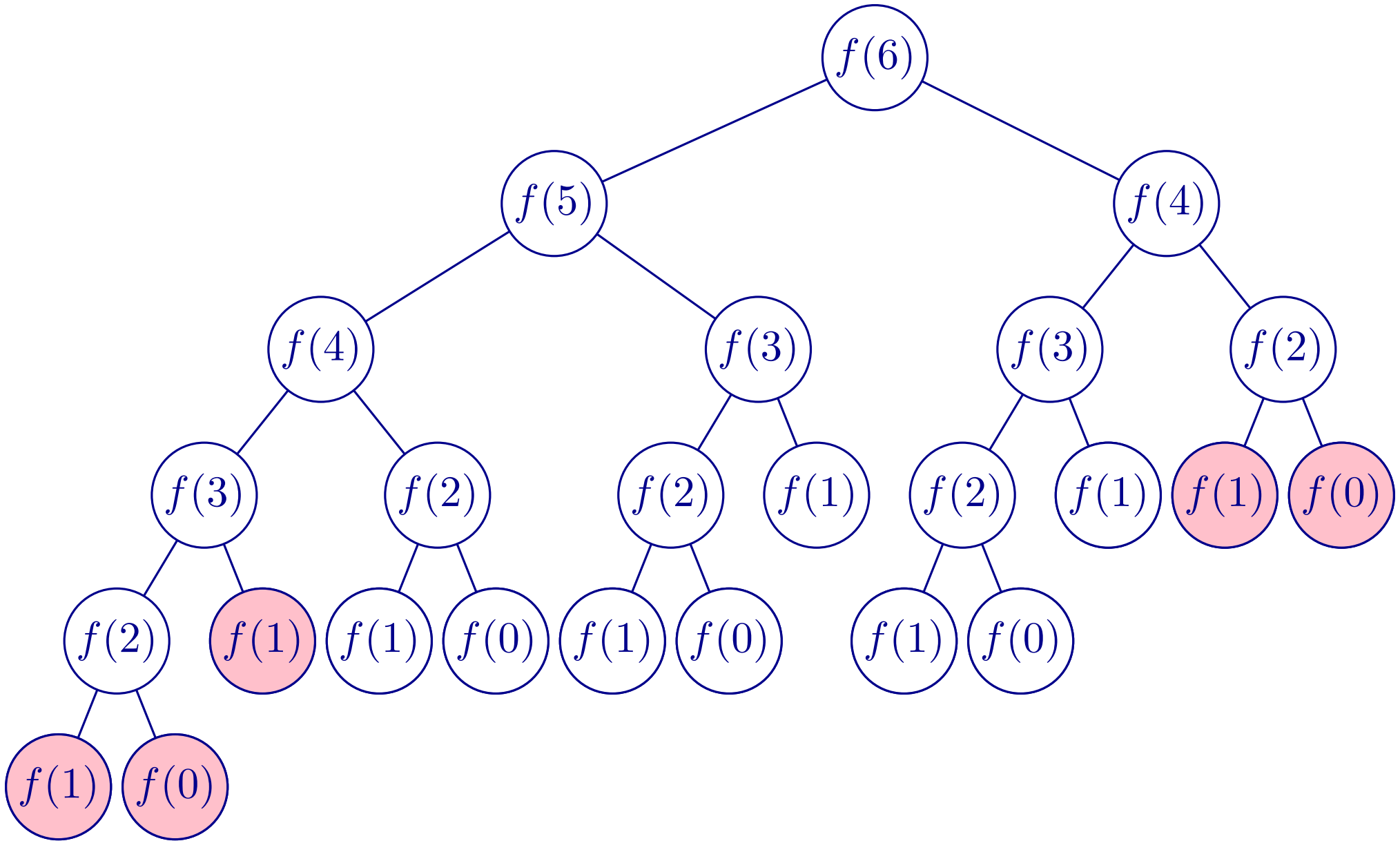
# Fibonacci Numbers: Time Complexity



# Fibonacci Numbers: Time Complexity



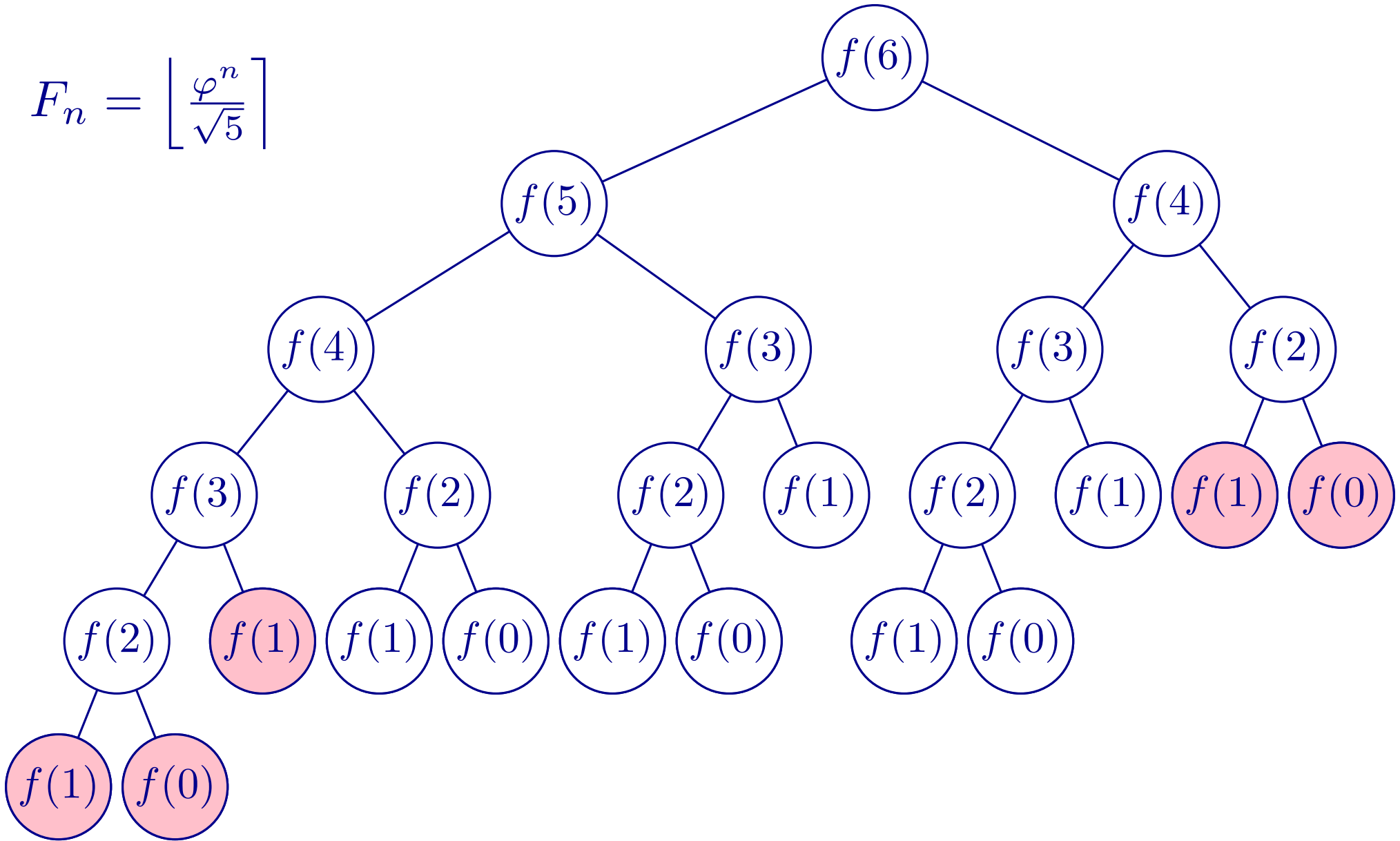
# Fibonacci Numbers: Time Complexity



$$\text{Time} = \Theta(1) \cdot \#\text{Nodes} = \Theta(\#\text{Leaves}) = \Theta(F_n)$$

# Fibonacci Numbers: Time Complexity

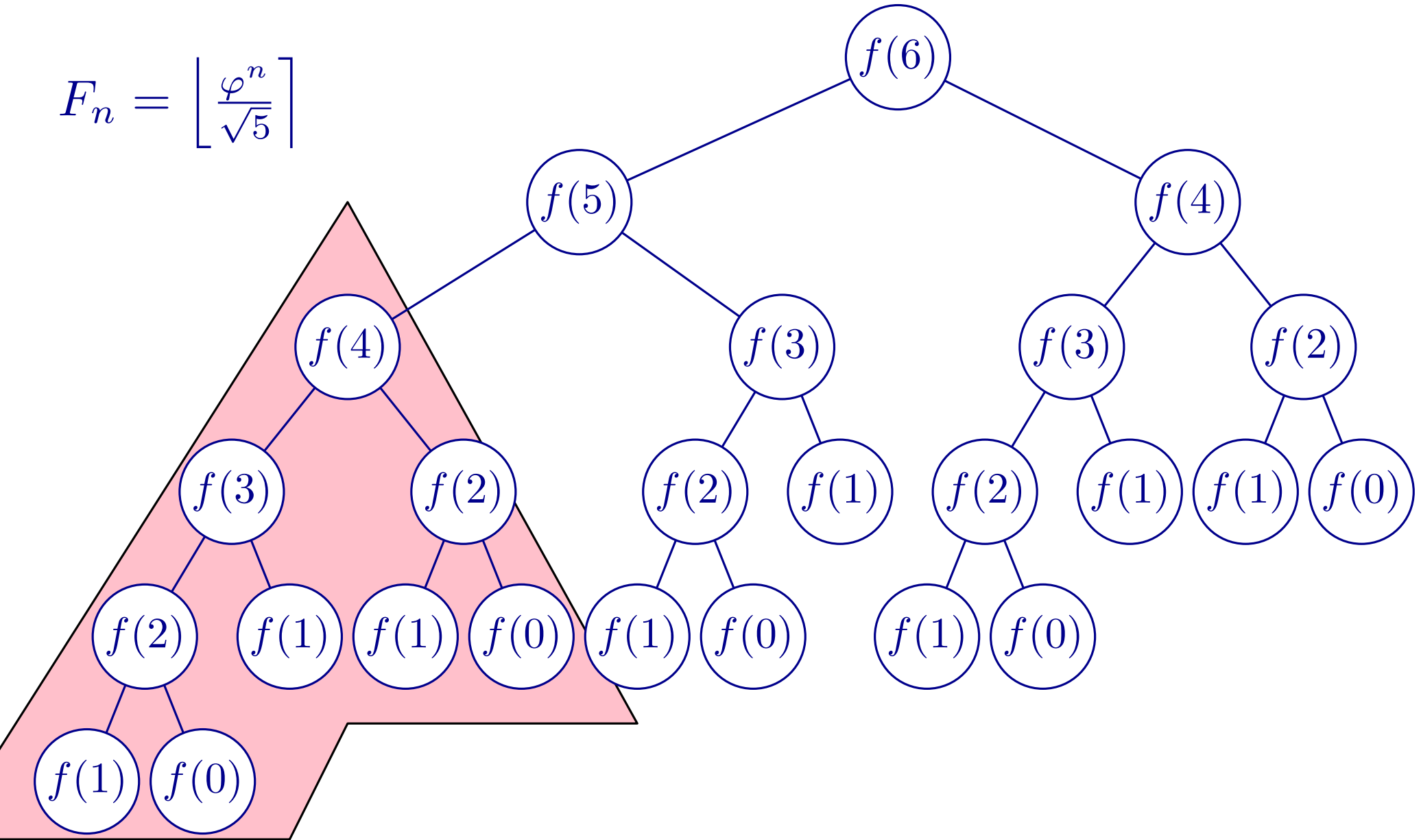
$$F_n = \left\lfloor \frac{\varphi^n}{\sqrt{5}} \right\rfloor$$



$$\text{Time} = \Theta(1) \cdot \#\text{Nodes} = \Theta(\#\text{Leaves}) = \Theta(F_n) = \Theta(\varphi^n)$$

# Fibonacci Numbers: Time Complexity

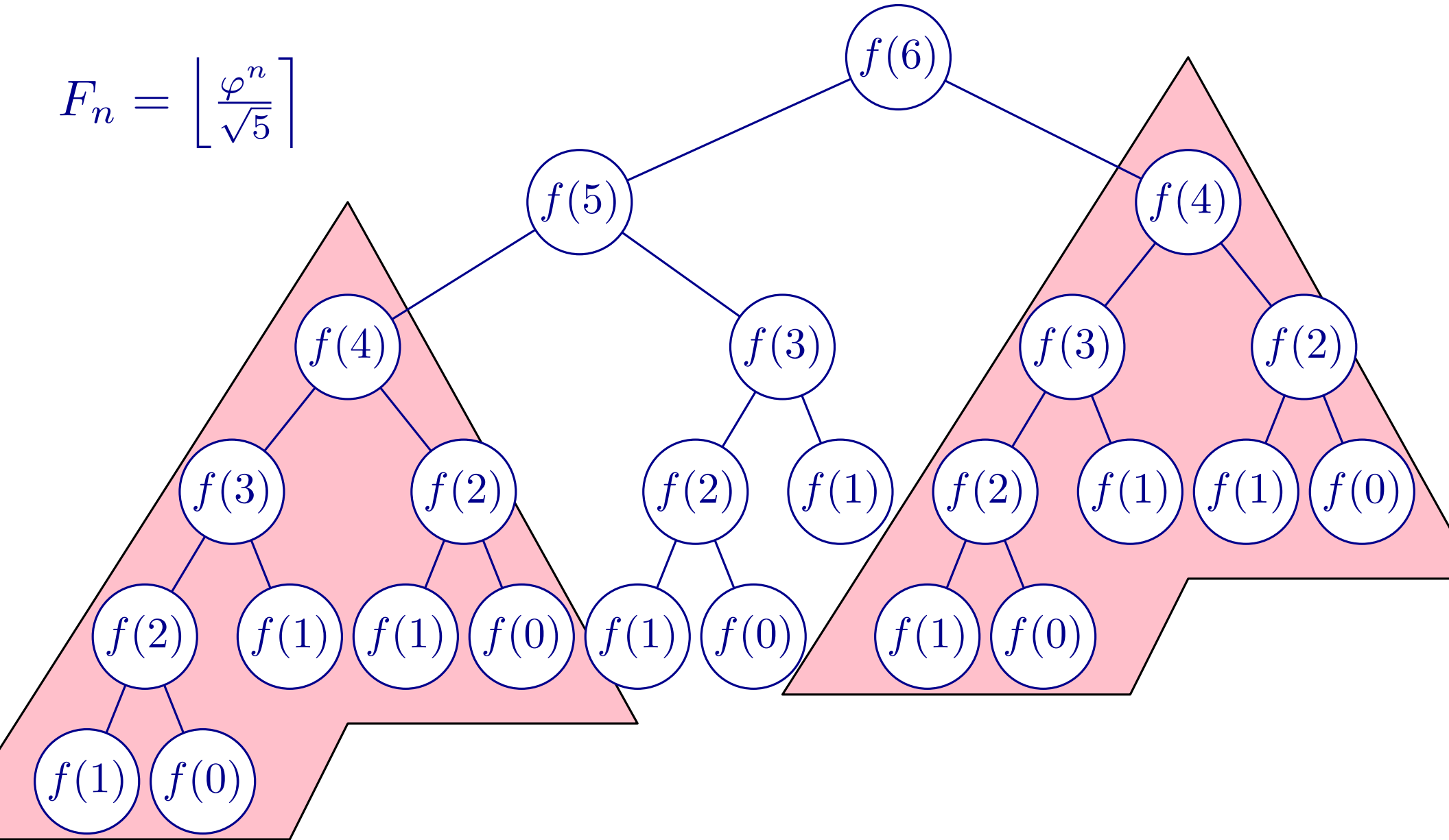
$$F_n = \left\lfloor \frac{\varphi^n}{\sqrt{5}} \right\rfloor$$



$$\text{Time} = \Theta(1) \cdot \# \text{Nodes} = \Theta(\# \text{Leaves}) = \Theta(F_n) = \Theta(\varphi^n)$$

# Fibonacci Numbers: Time Complexity

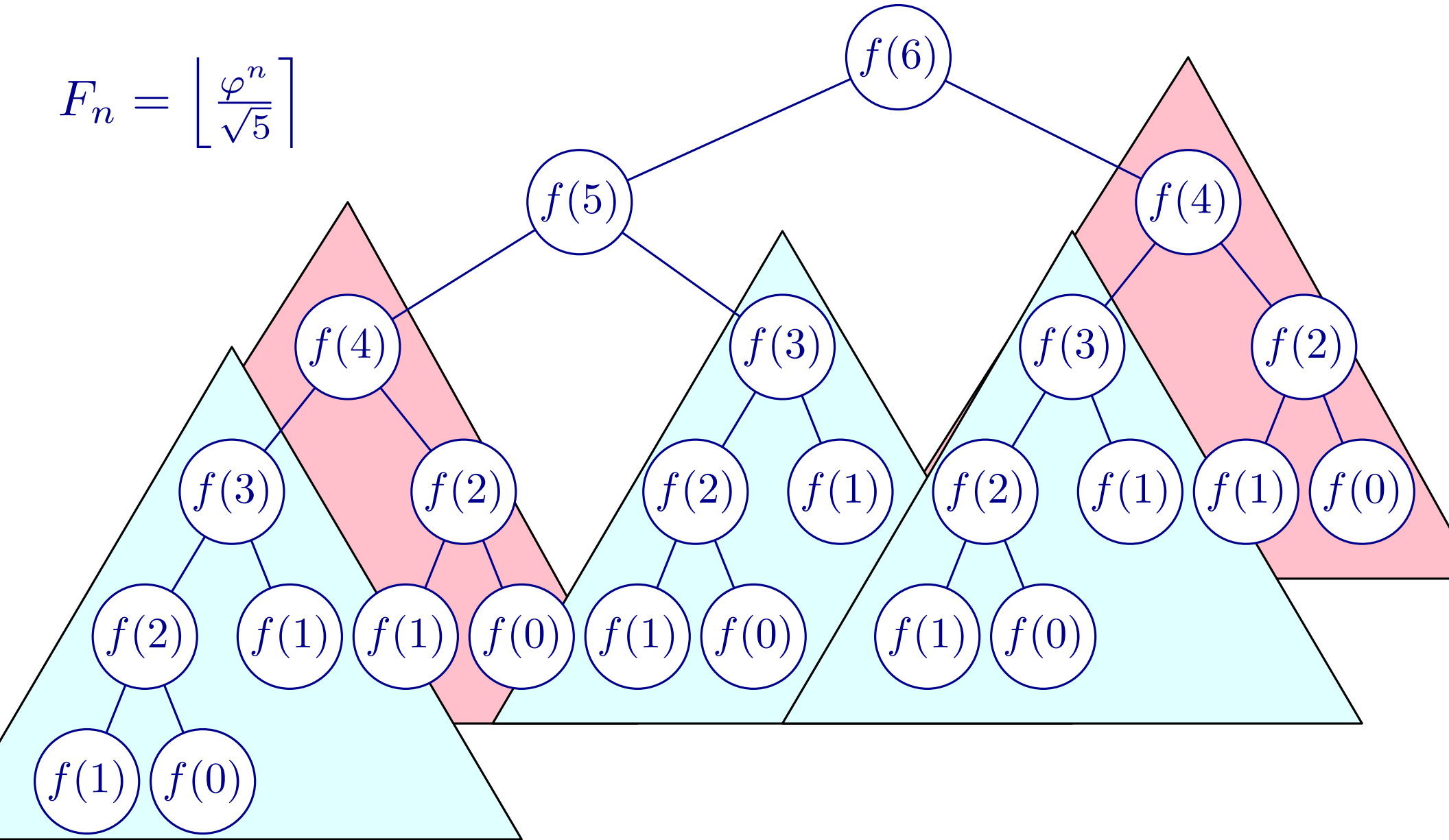
$$F_n = \left\lfloor \frac{\varphi^n}{\sqrt{5}} \right\rfloor$$



$$\text{Time} = \Theta(1) \cdot \# \text{Nodes} = \Theta(\# \text{Leaves}) = \Theta(F_n) = \Theta(\varphi^n)$$

# Fibonacci Numbers: Time Complexity

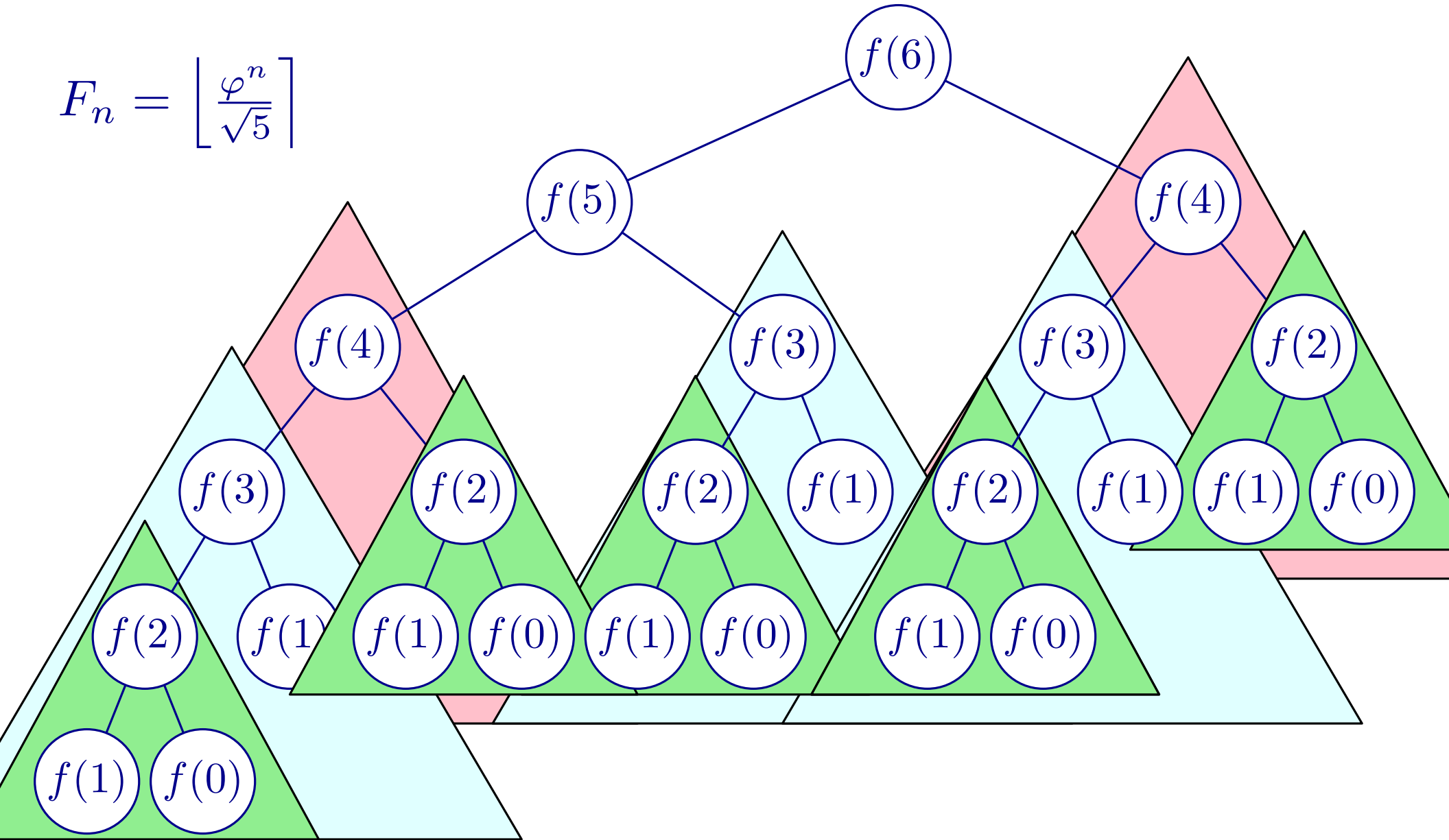
$$F_n = \left\lfloor \frac{\varphi^n}{\sqrt{5}} \right\rfloor$$



$$\text{Time} = \Theta(1) \cdot \#\text{Nodes} = \Theta(\#\text{Leaves}) = \Theta(F_n) = \Theta(\varphi^n)$$

# Fibonacci Numbers: Time Complexity

$$F_n = \left\lfloor \frac{\varphi^n}{\sqrt{5}} \right\rfloor$$



$$\text{Time} = \Theta(1) \cdot \# \text{Nodes} = \Theta(\# \text{Leaves}) = \Theta(F_n) = \Theta(\varphi^n)$$

# Fibonacci Numbers: Memoization

**Idea:** Do not recompute duplicate values:

- Store values in memory
- If value is in memory, recall it
- Otherwise, compute and store it

# Fibonacci Numbers: Memoization

**Idea:** Do not recompute duplicate values:

- Store values in memory
- If value is in memory, recall it
- Otherwise, compute and store it

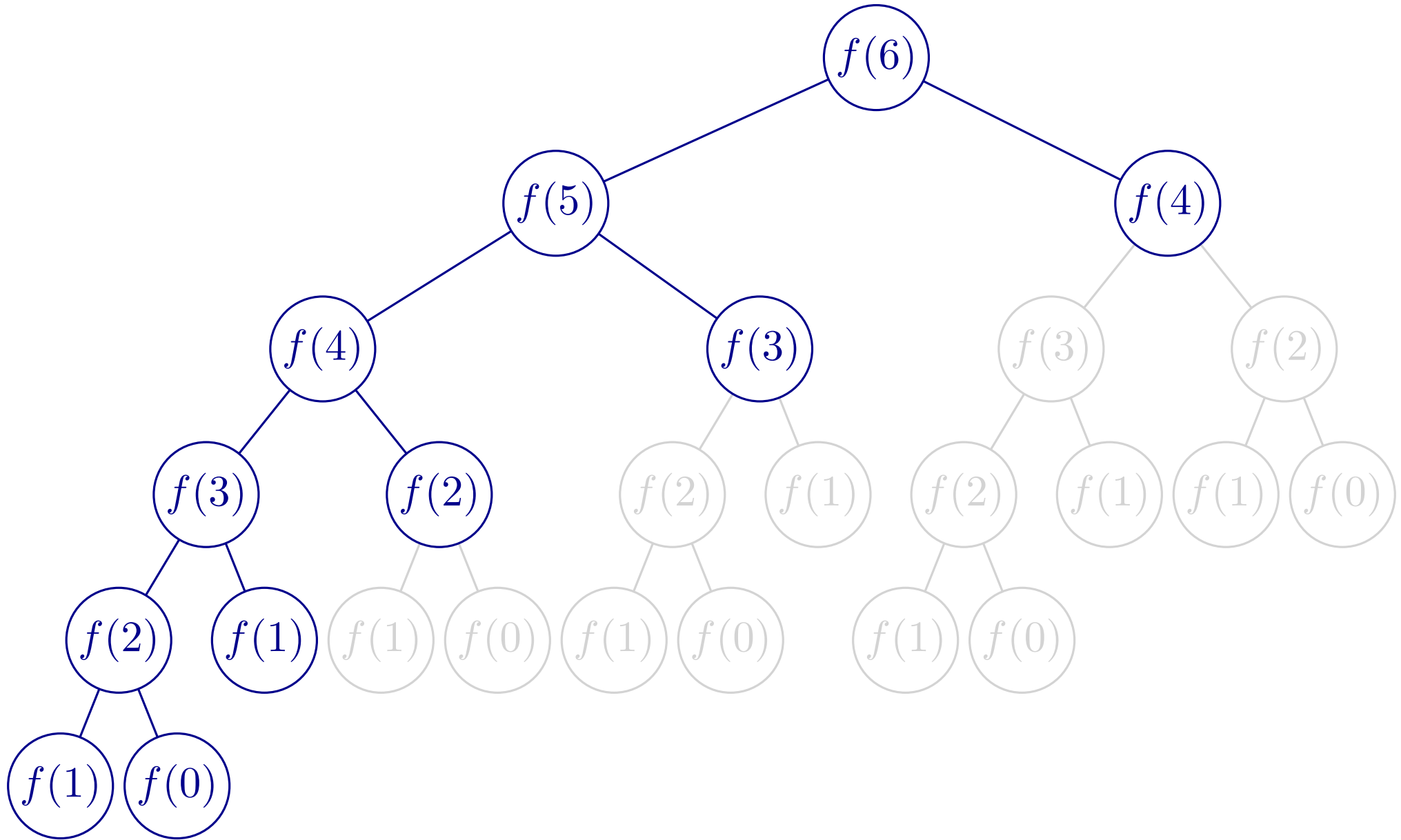
```
std::vector<int> memo(n+1, 0);

int fibonacci(int n)
{
    if(n<=1) return n;

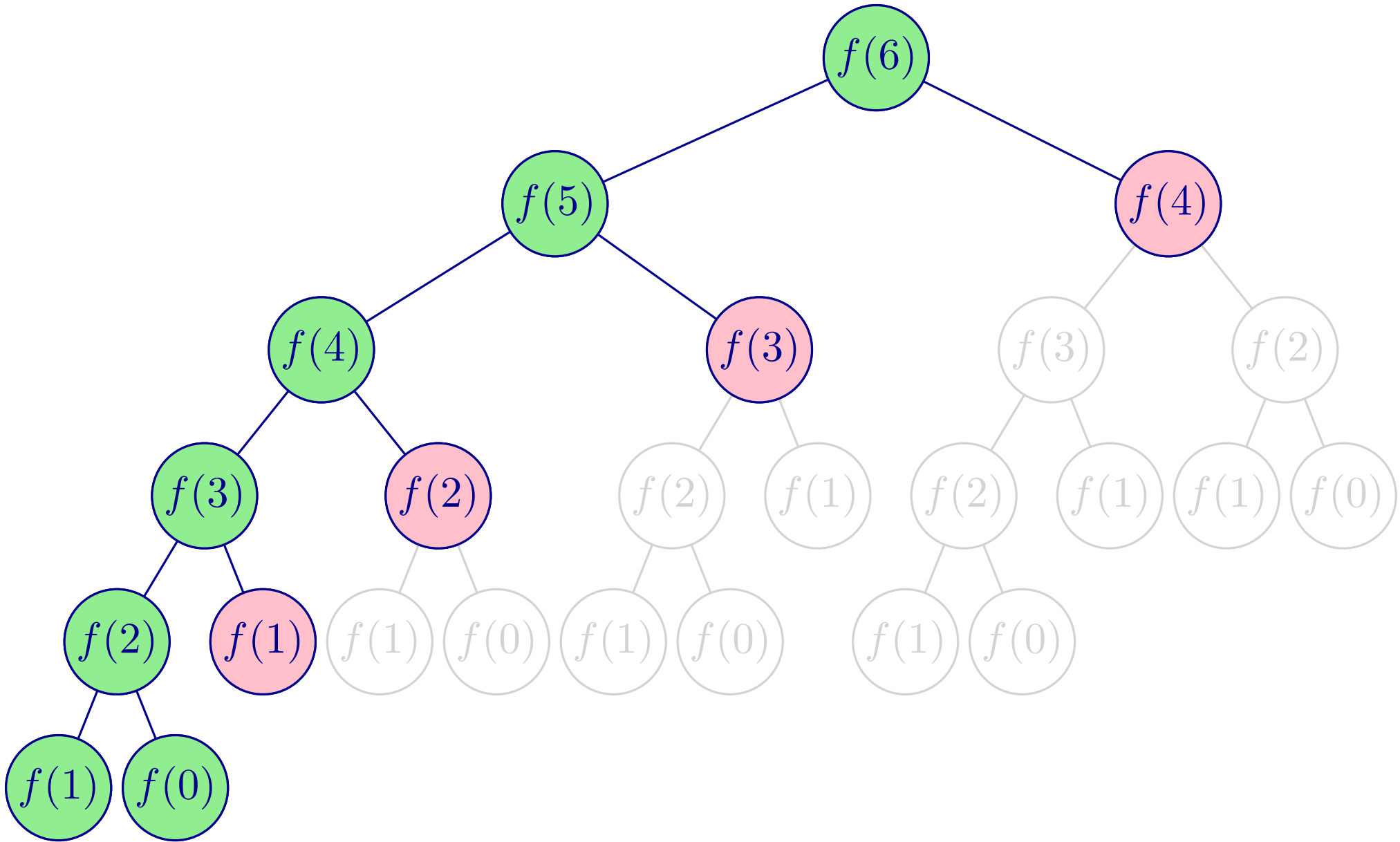
    if(memo[n]) return memo[n];

    memo[n] = fibonacci(n-1) + fibonacci(n-2);
    return memo[n];
}
```

# Time Complexity with Memoization



# Time Complexity with Memoization



$$\text{Time} = \Theta(1) \cdot \# \text{Green Nodes} = \Theta(n)$$

# The Memoization Recipe

- Design a recursive algorithm for the problem (hard)
- Add memoization (easy)

# The Memoization Recipe

- Design a recursive algorithm for the problem (hard)
- Add memoization (easy)
- Bound the computational complexity
  - How many subproblems (possible recursive calls)?
  - How long does a call take?

# The Memoization Recipe

- Design a recursive algorithm for the problem (hard)
- Add memoization (easy)
- Bound the computational complexity
  - How many subproblems (possible recursive calls)?
  - How long does a call take?

## Trick/Technique: Memoization

Avoid recomputing solutions to duplicate subproblems by storing results in memory.

# Memoization: Pitfalls

Let  $G_{-1} = G_0 = 1$ , and  $G_n = \begin{cases} 2G_{n-1} & \text{if } n \text{ is even} \\ G_{n-2} + 3 & \text{if } n \text{ is odd} \end{cases}$ , for  $n \geq 1$ .

```
std::vector<int> memo(n+1, 0);

int g(int n)
{
    if(memo[n]) return memo[n];

    if(n<=0) return 1;

    memo[n] = (n%2)?(g(n-2)+3):(2*g(n-1));
    return memo[n];
}
```

Does this code work?

# Memoization: Pitfalls

Let  $G_{-1} = G_0 = 1$ , and  $G_n = \begin{cases} 2G_{n-1} & \text{if } n \text{ is even} \\ G_{n-2} + 3 & \text{if } n \text{ is odd} \end{cases}$ , for  $n \geq 1$ .

```
std::vector<int> memo(n+1, 0);

int g(int n)
{
    if(memo[n]) return memo[n];

    if(n<=0) return 1;

    memo[n] = (n%2)?(g(n-2)+3):(2*g(n-1));
    return memo[n];
}
```



Does this code work?

# Memoization: Pitfalls

Let  $G_{-1} = G_0 = 1$ , and  $G_n = \begin{cases} 2G_{n-1} & \text{if } n \text{ is even} \\ G_{n-2} + 3 & \text{if } n \text{ is odd} \end{cases}$ , for  $n \geq 1$ .

```
std::vector<int> memo(n+1, 0);

int g(int n)
{
    if(memo[n]) return memo[n];

    if(n<=0) return 1;

    memo[n] = (n%2)?(g(n-2)+3):(2*g(n-1));
    return memo[n];
}
```



Does this code work? **No! n can be -1!**

# Memoization: Pitfalls

Let  $G_{-1} = G_0 = 1$ , and  $G_n = \begin{cases} 2G_{n-1} & \text{if } n \text{ is even} \\ G_{n-2} + 3 & \text{if } n \text{ is odd} \end{cases}$ , for  $n \geq 1$ .

```
std::vector<int> memo(n+1, 0);

int g(int n)
{
    if (memo[n]) return memo[n];
    if (n <= 0) return 1;
    memo[n] = (n%2)?(g(n-2)+3):(2*g(n-1));
    return memo[n];
}
```



Does this code work? **No!  $n$  can be  $-1$ !**

**Solution:** check base cases before the memo table.

# Memoization: Pitfalls

$G_0 = 0$ ,  $G_1 = 1$ , and  $G_n = (G_{n-1} + G_{n-2} + 1) \bmod 2$ , for  $n \geq 1$ .

```
std::vector<int> memo(n+1, 0);

int g(int n)
{
    if(n<=1) return n;

    if(memo[n]) return memo[n];

    memo[n] = (g(n-1) + g(n-2) + 1) % 2;
    return memo[n];
}
```

Too slow! Why?

# Memoization: Pitfalls

$G_0 = 0$ ,  $G_1 = 1$ , and  $G_n = (G_{n-1} + G_{n-2} + 1) \bmod 2$ , for  $n \geq 1$ .

```
std::vector<int> memo(n+1, 0);  
  
int g(int n)  
{  
    if(n<=1) return n;  
  
    if(memo[n]) return memo[n];  
  
    memo[n] = (g(n-1) + g(n-2) + 1) % 2;  
    return memo[n];  
}
```

Too slow! Why?

# Memoization: Pitfalls

$G_0 = 0$ ,  $G_1 = 1$ , and  $G_n = (G_{n-1} + G_{n-2} + 1) \bmod 2$ , for  $n \geq 1$ .

```
std::vector<int> memo(n+1, 0);  
  
int g(int n)  
{  
    if(n<=1) return n;  
  
    if(memo[n]) return memo[n];  
  
    memo[n] = (g(n-1) + g(n-2) + 1) % 2;  
    return memo[n];  
}
```

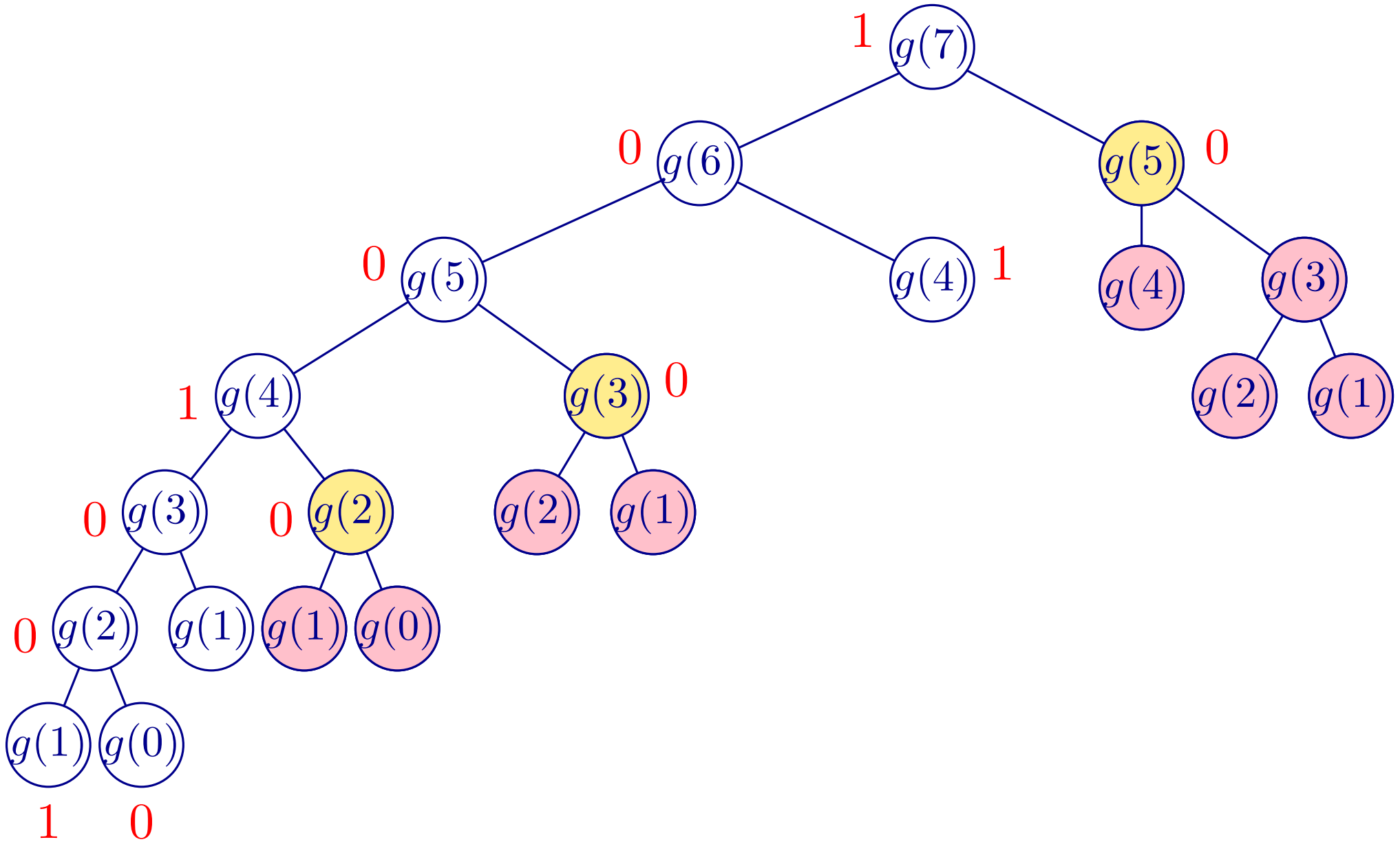
Too slow! Why?

0 is a possible value of  $G_n$  !



# Memoization: Pitfalls

$n = 7$



# Dynamic Programming

# Dynamic Programming

*I spent the Fall quarter (of 1950) at RAND. [...] We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. [...] he would get violent if people used the term research in his presence. [...] The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. [...] I decided therefore to use the word “programming”. I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. [...] Let’s take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it’s impossible to use the word dynamic in a pejorative sense. [...] Thus, I thought **dynamic programming** was a good name. It was something not even a Congressman could object to.*



Richard E. Bellman,  
Eye of the Hurricane: An Autobiography

# Dynamic Programming: Idea

- Decompose a problem into a series of “overlapping” subproblems
- The solutions to the “smallest” subproblems are trivially known
- The optimal solution to a subproblem can be reconstructed from the optimal solutions of “smaller” subproblems
- Systematically solve subproblems in a suitable order (from the “smaller” to the “larger” ones)
- Eventually, either the solution to the original problem is explicitly computed or it can be reconstructed from the subproblems’ solutions

# Dynamic Programming: Idea

- Decompose a problem into a series of “overlapping” subproblems (hard)
- The solutions to the “smallest” subproblems are trivially known (easy)
- The optimal solution to a subproblem can be reconstructed from the optimal solutions of “smaller” subproblems (hard)
- Systematically solve subproblems in a suitable order (from the “smaller” to the “larger” ones) (easy)
- Eventually, either the solution to the original problem is explicitly computed or it can be reconstructed from the subproblems’ solutions (easy)

# Fibonacci, Revisited

- $i$ -th subproblem: Compute the value of  $F_i$
- Base cases:  $i = 0, i = 1$ .
- Compute  $F_i$  in increasing order of  $i$ :  $F_i = F_{i-1} + F_{i-2}$
- Both  $F_{i-1}$  and  $F_{i-2}$  are already known when  $F_i$  is considered.
- Solution:  $F_n$

```
std::vector<int> F(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];
```

# Fibonacci, Revisited

Trick to reduce space:

- Once we compute  $F_i$ , the values  $F_0, \dots, F_{i-2}$  will not be used anymore.
- Keep track of just two values  $x_0, x_1$ .
- At the end of iteration  $i$ ,  $F_i = x_{i \bmod 2}$  and  $F_{i-1} = x_{(i-1) \bmod 2}$ .

```
int x[2] = {0, 1};

for(int i=2; i<=n; i++)
    x[i%2] = x[(i-1)%2] + x[(i-2)%2];

return x[n%2];
```

# Fibonacci, Revisited

Trick to reduce space:

- Once we compute  $F_i$ , the values  $F_0, \dots, F_{i-2}$  will not be used anymore.
- Keep track of just two values  $x_0, x_1$ .
- At the end of iteration  $i$ ,  $F_i = x_{i \bmod 2}$  and  $F_{i-1} = x_{(i-1) \bmod 2}$ .

```
int x[2] = {0, 1};  
for(int i=2; i<=n; i++)  
    x[i%2] = x[(i-1)%2] + x[(i-2)%2];  
return x[n%2];
```

$F_i$

$F_{i-1}$

$F_{i-2}$

# Drink as much as possible

Robert wants to drink as much as possible.

- Robert walks through the streets of King's Landing and encounters  $n$  taverns  $t_1, t_2, \dots, t_n$ , in order
- When Robert encounters a tavern  $t_i$ , he can either stop for a drink or continue walking
- The wine served in tavern  $t_i$  has strength  $s_i \in \mathbb{N}$  (the higher, the stronger)
- The strength of Robert's drinks must increase over time
- **Goal:** Compute the maximum number of drinking stops of Robert

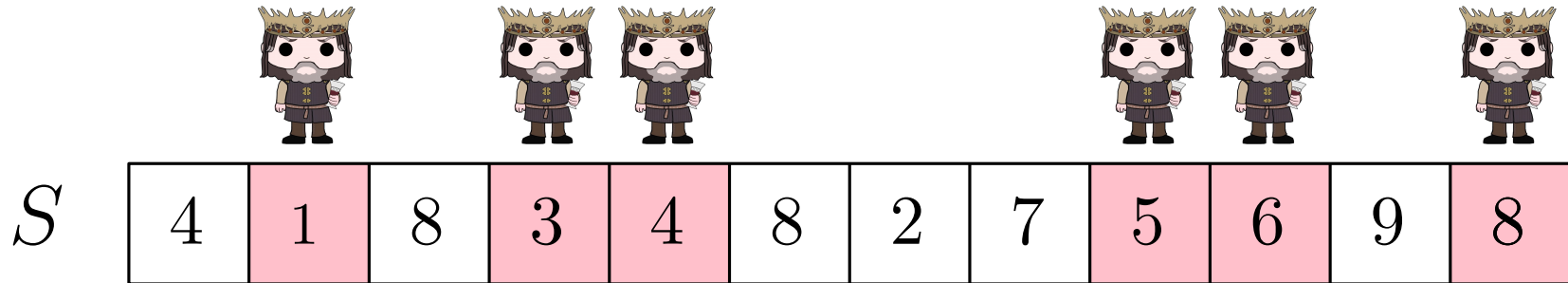


# Example

*S*

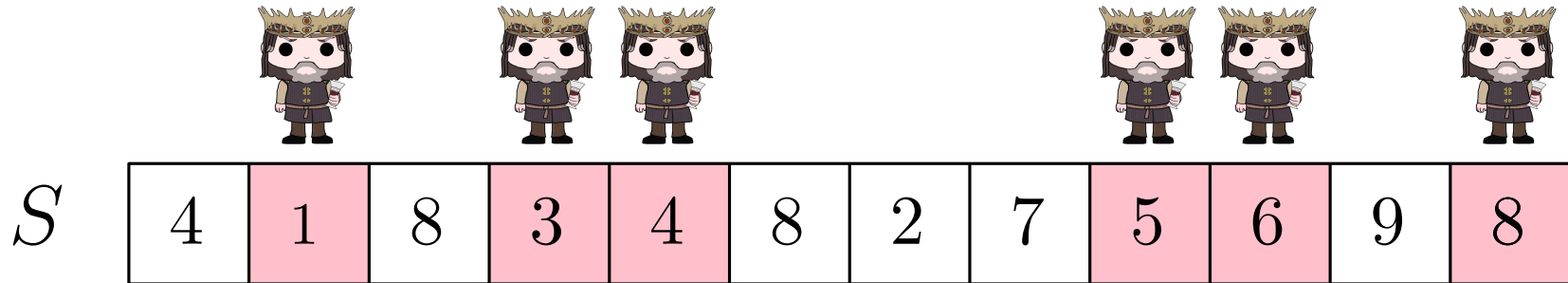
|   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 1 | 8 | 3 | 4 | 8 | 2 | 7 | 5 | 6 | 9 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|---|

# Example



**Solution: 6**

# Example



**Solution: 6**

This is a classic problem known as:  
**Longest Increasing Subsequence (LIS)**

# A DP Algorithm: First Attempt

- Subproblem definition

$$OPT[i] = \text{Length of the LIS in } S[1], \dots, S[i]$$

# A DP Algorithm: First Attempt

- Subproblem definition

$$OPT[i] = \text{Length of the LIS in } S[1], \dots, S[i]$$

- Base cases

$$OPT[1] = 1$$

# A DP Algorithm: First Attempt

- Subproblem definition

$$OPT[i] = \text{Length of the LIS in } S[1], \dots, S[i]$$

- Base cases

$$OPT[1] = 1$$

- Solution:

$$OPT[n]$$

# A DP Algorithm: First Attempt

- Subproblem definition

$$OPT[i] = \text{Length of the LIS in } S[1], \dots, S[i]$$

- Base cases

$$OPT[1] = 1$$

- Solution:

$$OPT[n]$$

- Recursive formula



# A DP Algorithm: Second Attempt

**Tip:** Sometimes adding constraints to subproblems helps!

# A DP Algorithm: Second Attempt

**Tip:** Sometimes adding constraints to subproblems helps!

$OPT[i]$  = Length of the LIS that ends with  $S[i]$

# A DP Algorithm: Second Attempt

**Tip:** Sometimes adding constraints to subproblems helps!

$OPT[i]$  = Length of the LIS that ends with  $S[i]$

|       |   |   |   |   |   |   |   |   |   |    |    |    |
|-------|---|---|---|---|---|---|---|---|---|----|----|----|
| $S$   | 4 | 1 | 8 | 3 | 4 | 8 | 2 | 7 | 5 | 6  | 9  | 8  |
|       | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| $OPT$ | 1 | 1 | 2 | 2 | 3 | 4 | 2 | 4 |   |    |    |    |

# A DP Algorithm: Second Attempt

**Tip:** Sometimes adding constraints to subproblems helps!


$OPT[i] =$  Length of the LIS that ends with  $S[i]$

|       |   |   |   |   |   |   |   |   |   |    |    |    |
|-------|---|---|---|---|---|---|---|---|---|----|----|----|
| $S$   | 4 | 1 | 8 | 3 | 4 | 8 | 2 | 7 | 5 | 6  | 9  | 8  |
|       | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| $OPT$ | 1 | 1 | 2 | 2 | 3 | 4 | 2 | 4 |   |    |    |    |

# A DP Algorithm: Second Attempt

**Tip:** Sometimes adding constraints to subproblems helps!

$OPT[i] =$  Length of the LIS that ends with  $S[i]$




|       |   |   |   |   |   |   |   |   |   |    |    |    |
|-------|---|---|---|---|---|---|---|---|---|----|----|----|
| $S$   | 4 | 1 | 8 | 3 | 4 | 8 | 2 | 7 | 5 | 6  | 9  | 8  |
|       | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| $OPT$ | 1 | 1 | 2 | 2 | 3 | 4 | 2 | 4 |   |    |    |    |

# A DP Algorithm: Second Attempt

**Tip:** Sometimes adding constraints to subproblems helps!

$OPT[i]$  = Length of the LIS that ends with  $S[i]$




|       |   |   |   |   |   |   |   |   |   |    |    |    |
|-------|---|---|---|---|---|---|---|---|---|----|----|----|
| $S$   | 4 | 1 | 8 | 3 | 4 | 8 | 2 | 7 | 5 | 6  | 9  | 8  |
|       | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| $OPT$ | 1 | 1 | 2 | 2 | 3 | 4 | 2 | 4 |   |    |    |    |

**Possible lengths: 3**

# A DP Algorithm: Second Attempt

**Tip:** Sometimes adding constraints to subproblems helps!

$OPT[i] =$  Length of the LIS that ends with  $S[i]$




|       |   |   |   |   |   |   |   |   |   |    |    |    |
|-------|---|---|---|---|---|---|---|---|---|----|----|----|
| $S$   | 4 | 1 | 8 | 3 | 4 | 8 | 2 | 7 | 5 | 6  | 9  | 8  |
|       | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| $OPT$ | 1 | 1 | 2 | 2 | 3 | 4 | 2 | 4 |   |    |    |    |

**Possible lengths: 3**

# A DP Algorithm: Second Attempt

**Tip:** Sometimes adding constraints to subproblems helps!

$OPT[i] =$  Length of the LIS that ends with  $S[i]$



|       |   |   |   |   |   |   |   |   |   |    |    |    |
|-------|---|---|---|---|---|---|---|---|---|----|----|----|
| $S$   | 4 | 1 | 8 | 3 | 4 | 8 | 2 | 7 | 5 | 6  | 9  | 8  |
|       | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| $OPT$ | 1 | 1 | 2 | 2 | 3 | 4 | 2 | 4 |   |    |    |    |

**Possible lengths:** 3 4

# A DP Algorithm: Second Attempt

**Tip:** Sometimes adding constraints to subproblems helps!

$OPT[i] =$  Length of the LIS that ends with  $S[i]$



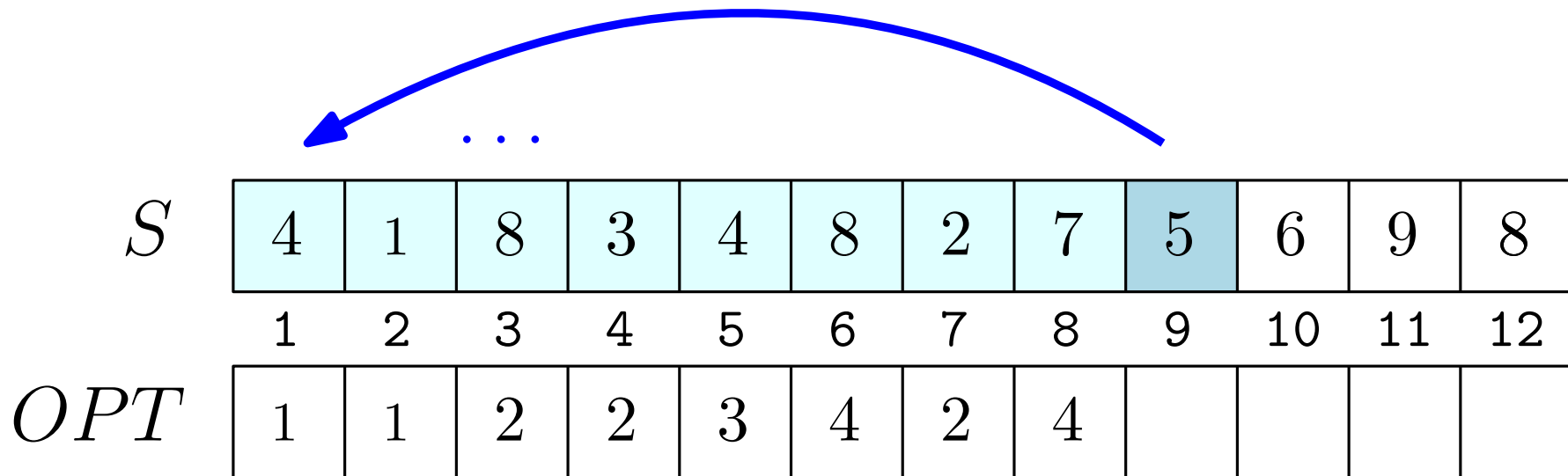
|       |   |   |   |   |   |   |   |   |   |    |    |    |
|-------|---|---|---|---|---|---|---|---|---|----|----|----|
| $S$   | 4 | 1 | 8 | 3 | 4 | 8 | 2 | 7 | 5 | 6  | 9  | 8  |
|       | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| $OPT$ | 1 | 1 | 2 | 2 | 3 | 4 | 2 | 4 |   |    |    |    |

**Possible lengths:** 3 4 3

# A DP Algorithm: Second Attempt

**Tip:** Sometimes adding constraints to subproblems helps!

$OPT[i] =$  Length of the LIS that ends with  $S[i]$

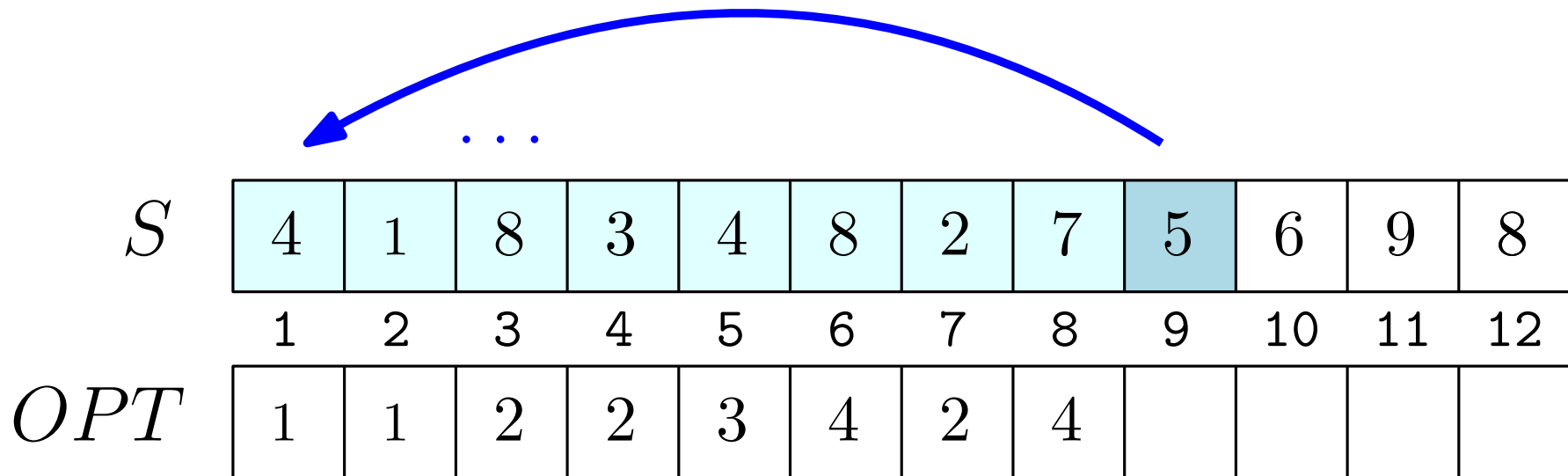


**Possible lengths:** 3 4 3 2 2

# A DP Algorithm: Second Attempt

**Tip:** Sometimes adding constraints to subproblems helps!

$OPT[i]$  = Length of the LIS that ends with  $S[i]$



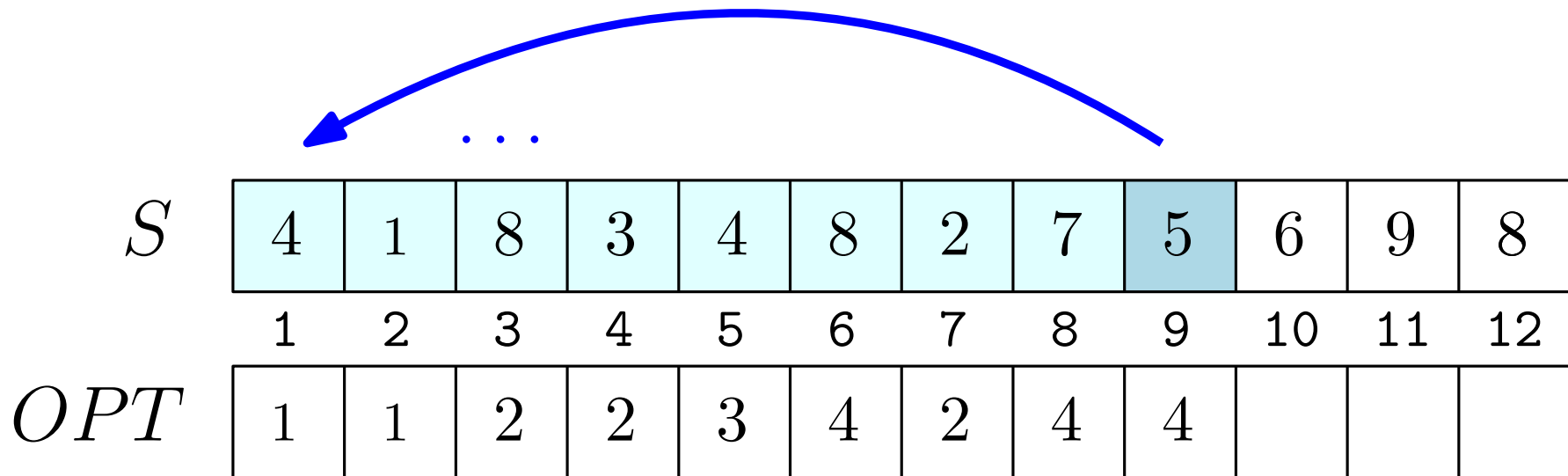
**Possible lengths:** 3 4 3 2 2 1

Sequence containing only  $S[i]$

# A DP Algorithm: Second Attempt

**Tip:** Sometimes adding constraints to subproblems helps!

$OPT[i] =$  Length of the LIS that ends with  $S[i]$



**Possible lengths:** 3 4 3 2 2 1       $OPT[9] = 4$

Sequence containing only  $S[i]$

# The Dynamic Programming Algorithm

- Subproblem definition

$$OPT[i] = \text{Length of the LIS that ends with } S[i]$$

- Base cases

$$OPT[1] = 1$$

- Recursive formula (for  $i \geq 2$ )

$$OPT[i] = \max \left\{ 1, 1 + \max_{\substack{j=1, \dots, i-1 \\ S[j] < S[i]}} OPT[j] \right\}$$

- Subproblems' order

$$OPT[1], OPT[2], \dots, OPT[n]$$

- Solution:

$$\max_{i=1, \dots, n} OPT[i]$$

# Time Complexity

- $O(n)$  subproblems
- Base cases are handled in constant time
- $OPT[i]$  is computed in time  $\Theta(i)$

$$OPT[i] = \max \left\{ 1, 1 + \max_{\substack{j=1, \dots, i-1 \\ S[j] < S[i]}} OPT[j] \right\}$$

# Time Complexity

- $O(n)$  subproblems
- Base cases are handled in constant time
- $OPT[i]$  is computed in time  $\Theta(i)$

$$OPT[i] = \max \left\{ 1, 1 + \max_{\substack{j=1, \dots, i-1 \\ S[j] < S[i]}} OPT[j] \right\}$$

**Overall time:**  $O\left(\sum_{i=1}^n i\right) = O(n^2)$ .

# Time Complexity

- $O(n)$  subproblems
- Base cases are handled in constant time
- $OPT[i]$  is computed in time  $\Theta(i)$

$$OPT[i] = \max \left\{ 1, 1 + \max_{\substack{j=1, \dots, i-1 \\ S[j] < S[i]}} OPT[j] \right\}$$

**Overall time:**  $O\left(\sum_{i=1}^n i\right) = O(n^2)$ .

There is also another dynamic-programming algorithm for the LIS problem running in time  $O(n \log n)$  [Fredman, 1985]

# A possible implementation (DP)

```
std::vector<int> OPT(n+1);
OPT[1]=1;

for(int i=2; i<=n; i++)
{
    OPT[i]=1;
    for(int j=1; j<i; j++)
        if(S[j] < S[i])
            OPT[i] = std::max(OPT[i], 1+OPT[j]);
}

return std::max_element(OPT.begin()+1, OPT.end());
```

# A possible implementation (Memo)

```
std::vector<int> memo(n+1, 0);

int LIS(std::vector &S, int i)
{
    if(i==1) return 1;

    if(memo[i]) return memo[i];

    int r=1;
    for(int j=1; j<i; j++)
        if(S[j]<S[i])
            r=std::max(r, 1+LIS(S, j));

    return memo[i]=r;
}
```

# Memoization vs. DP

- ✓ Top-Down approach (more intuitive)
- ✓ Easier to index subproblems by other objects (e.g., sets).
- ✓ Only computes necessary subproblems
- ✗ Function calls overhead
- ✗ Call stack (recursion depth) is bounded
- ✗ Time complexity is harder to analyze

- ✗ Bottom-Up approach (harder to grasp)
- ✗ Need to index subproblems with integers
- ✗ Always computes all subproblems
- ✓ No recursion. Less overhead. More cache efficient.
- ✓ Short and clean code
- ✓ Time complexity analysis is easy(/ier)

Recap

## Trick/Technique: Divide and Conquer

Decompose an instance into smaller instances of the same problem.

Solve recursively and recombine the solutions.

## **Trick/Technique: Divide and Conquer**

Decompose an instance into smaller instances of the same problem.

Solve recursively and recombine the solutions.

## **Trick/Technique: Memoization**

Avoid recomputing solutions to duplicate subproblems by storing results in memory.

## **Trick/Technique: Divide and Conquer**

Decompose an instance into smaller instances of the same problem.

Solve recursively and recombine the solutions.

## **Trick/Technique: Memoization**

Avoid recomputing solutions to duplicate subproblems by storing results in memory.

## **Trick/Technique: Dynamic Programming**

Define overlapping subproblems (possibly w/additional constraints). Systematically solve subproblems using an order that allows previous solutions to be recombined. Compute solution to the original problem from the subproblems' solutions.