# CSE- 207
# Algorithms

**Lecture: 11**

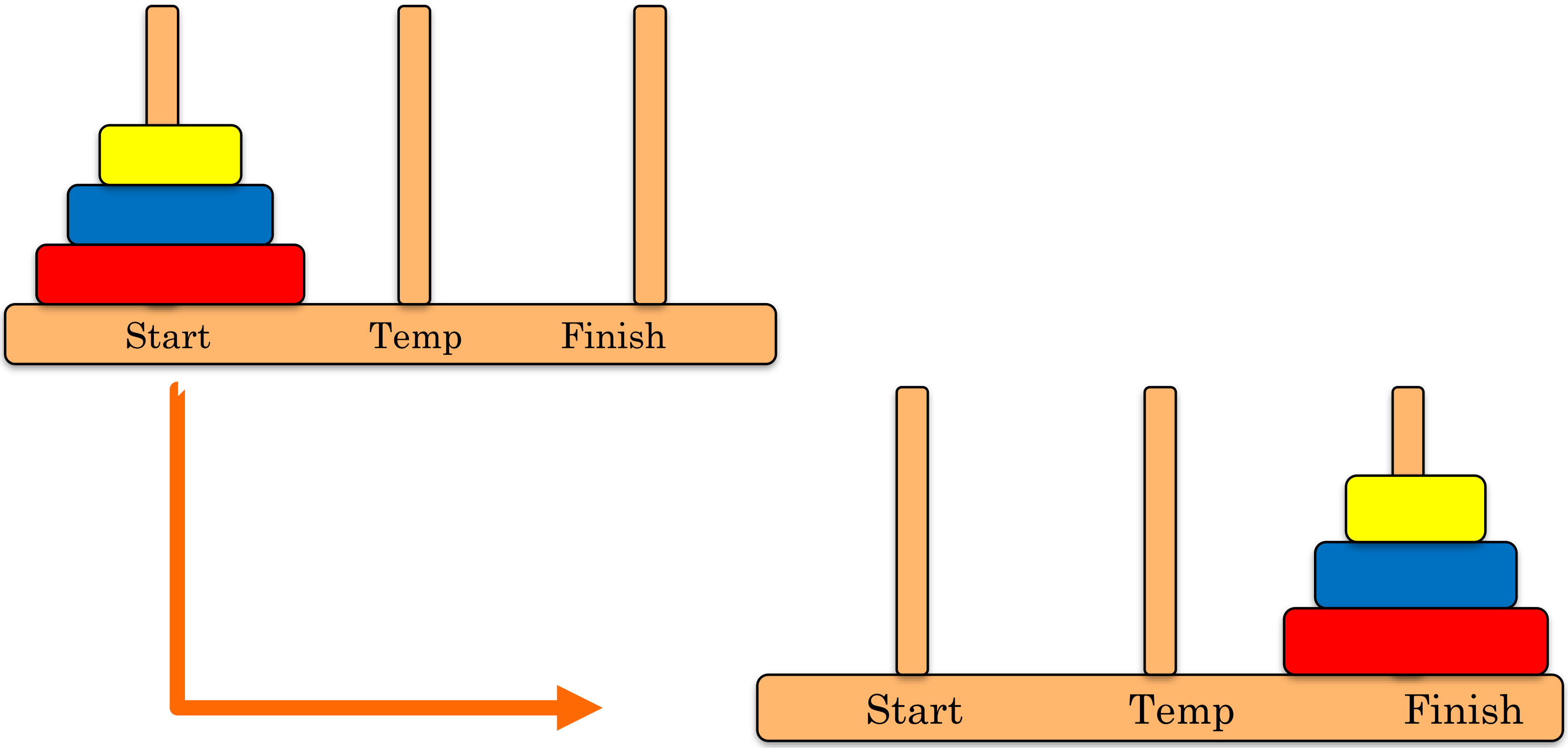
Recursion and
Dynamic Programming

**Fahad Ahmed**

Lecturer, Dept. of CSE

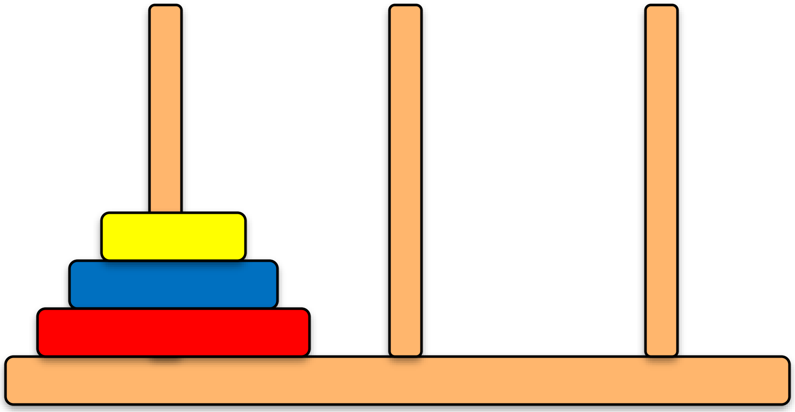
E-mail: fahadahmed@uap-bd.edu

# Towers of Hanoi



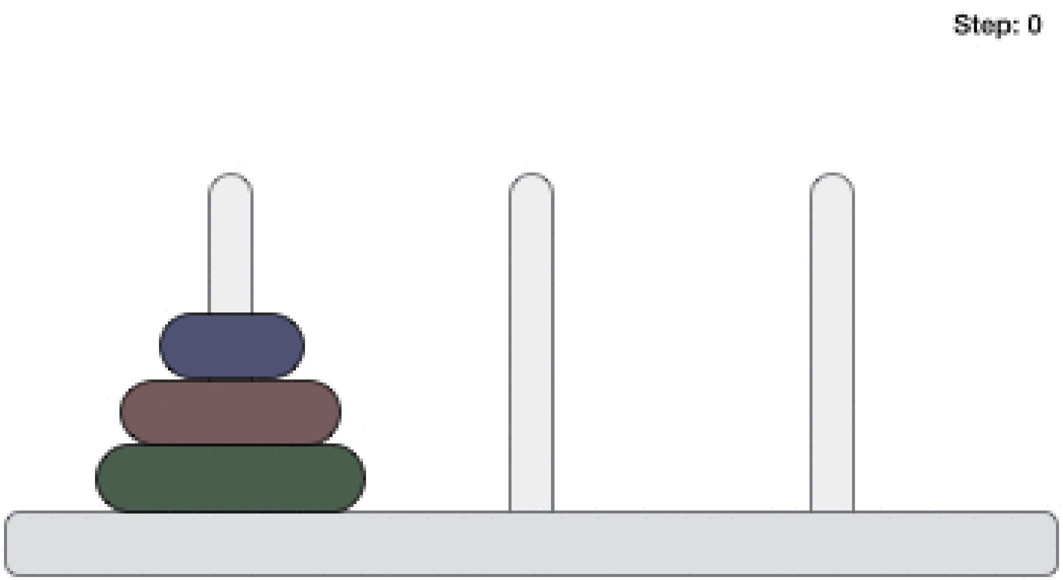
# Recursion – Towers of Hanoi

- The Towers of Hanoi
  - Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted
  - These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one.
  - There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.



# Recursion – Towers of Hanoi

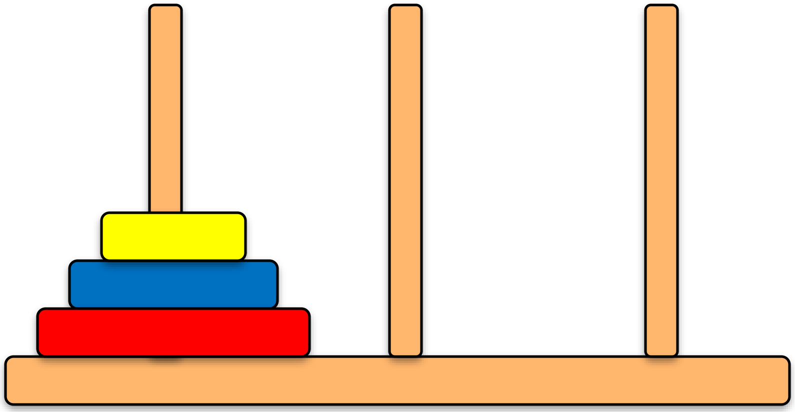
- The Towers of Hanoi
  - The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −
    - Only one disk can be moved among the towers at any given time.
    - Only the "top" disk can be removed.
    - No large disk can sit over a small disk.



# Recursion – Towers of Hanoi

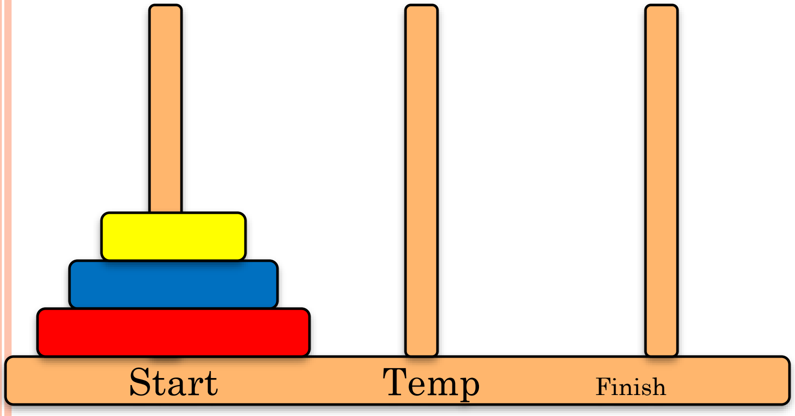
- The Towers of Hanoi
  - Solution:

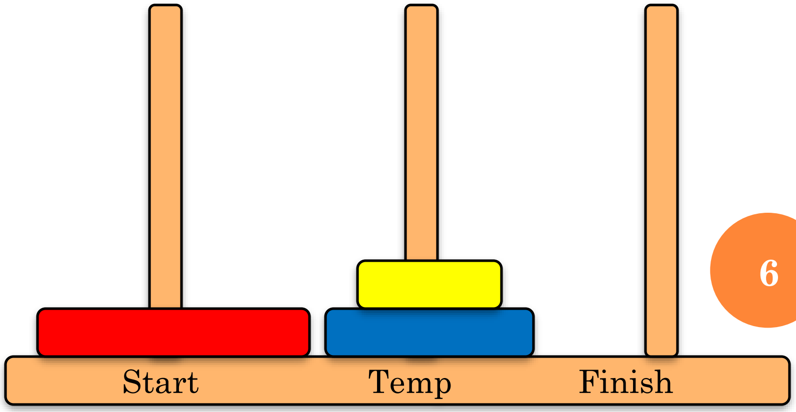
1) The steps to follow are –
2) Step 1 – Move n-1 disks from source to aux
3) Step 2 – Move $n^{th}$ disk from source to dest
4) Step 3 – Move n-1 disks from aux to dest



# Recursion – Towers of Hanoi

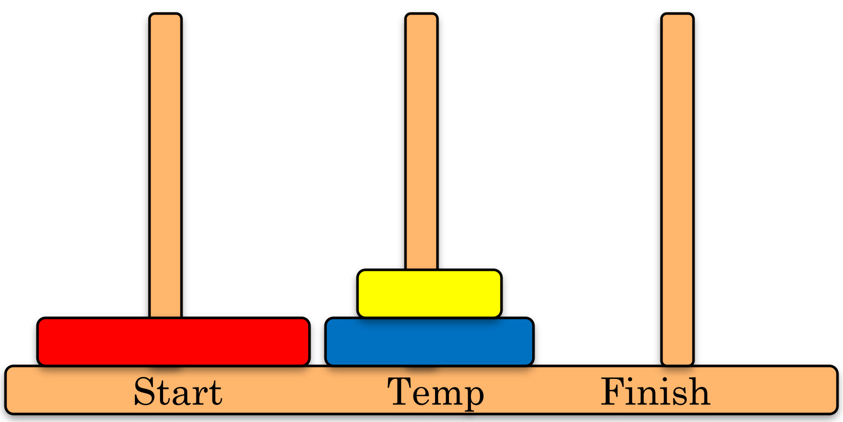
- The Towers of Hanoi
  - Let's look at the problem with only 3 disks.
  - Solution:
    - Step 1:
      - Move top 2 disks to temp
        - we would have to solve this recursively, since we can only move 2 disks at a time.
        - We're going to assume that we know how to do the 2 disk problem (since this is solved recursively), and continue to the next step.

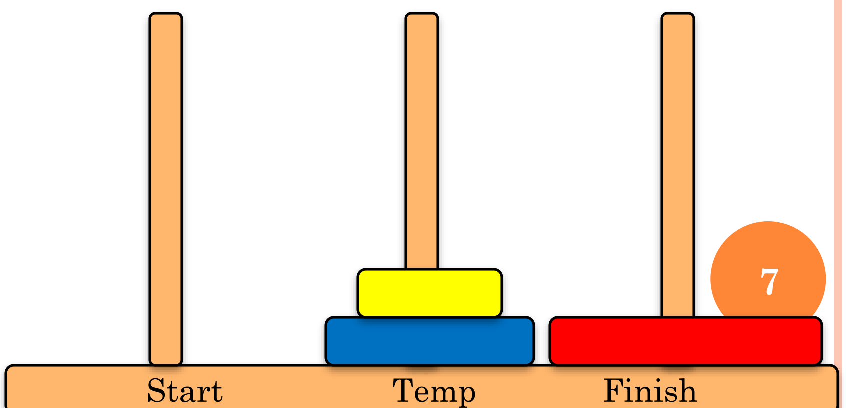


# Recursion – Towers of Hanoi

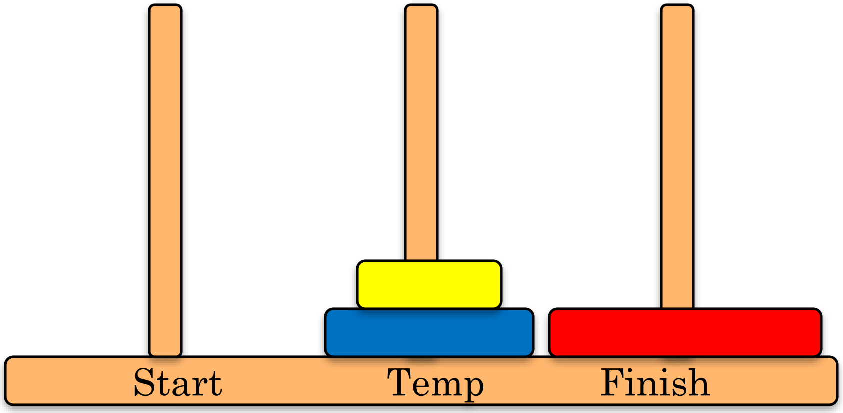
- The Towers of Hanoi
  - Let's look at the problem with only 3 disks.
  - Solution:
    - Step 2:
      - Move the last single disk from start to finish
      - Moving a single disk does not use recursion, and does not use the temp tower.
      - (In our program, a single disk move is represented with a print statement.)

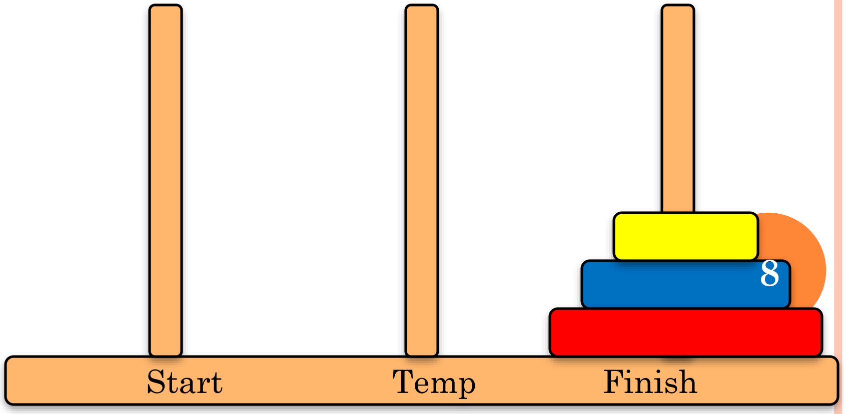


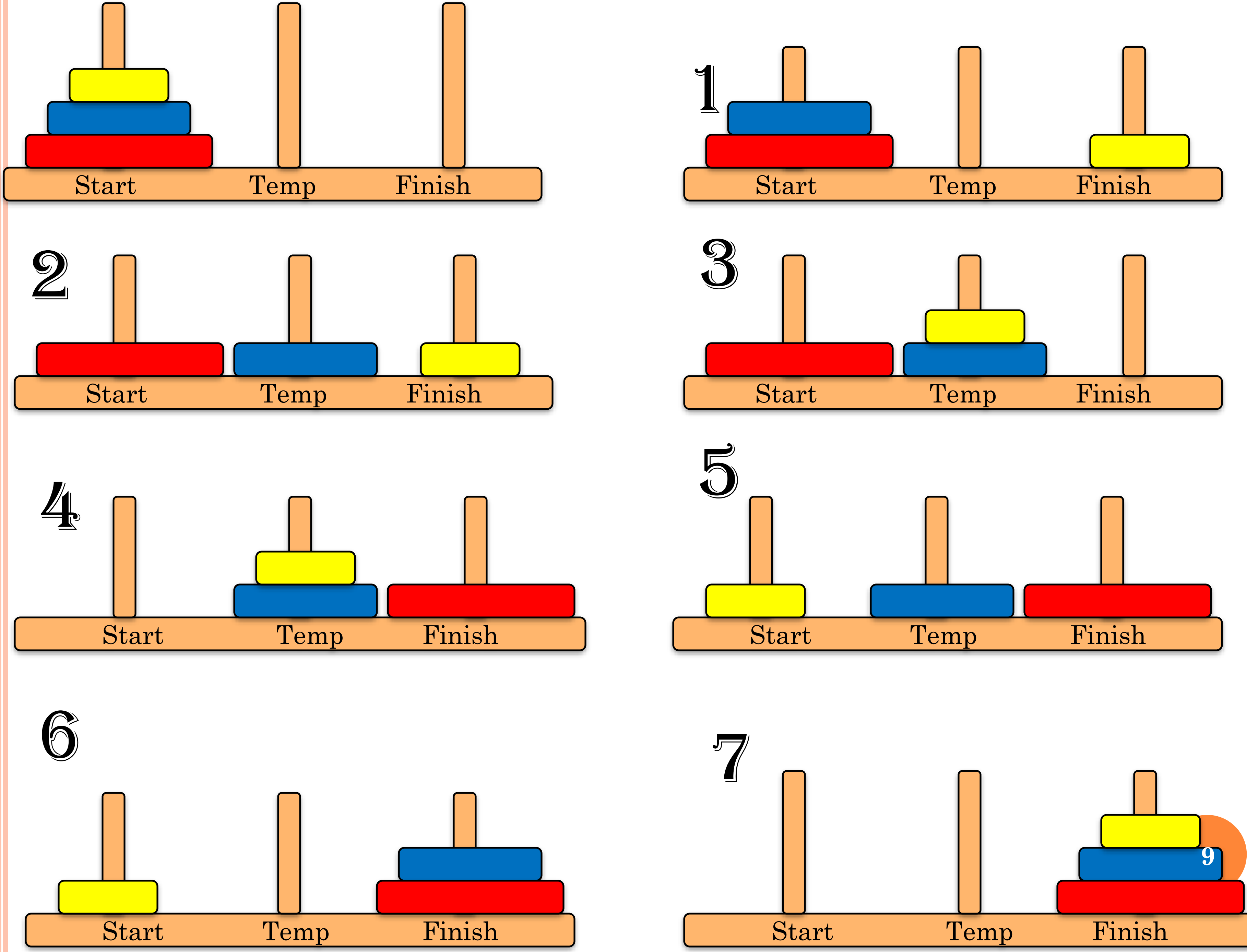



# Recursion – Towers of Hanoi

- The Towers of Hanoi
  - Let's look at the problem with only 3 disks.
  - Solution:
    - Step 3:
      - Last step – Move the 2 disks from Temp to Finish
        - This would be done recursively.

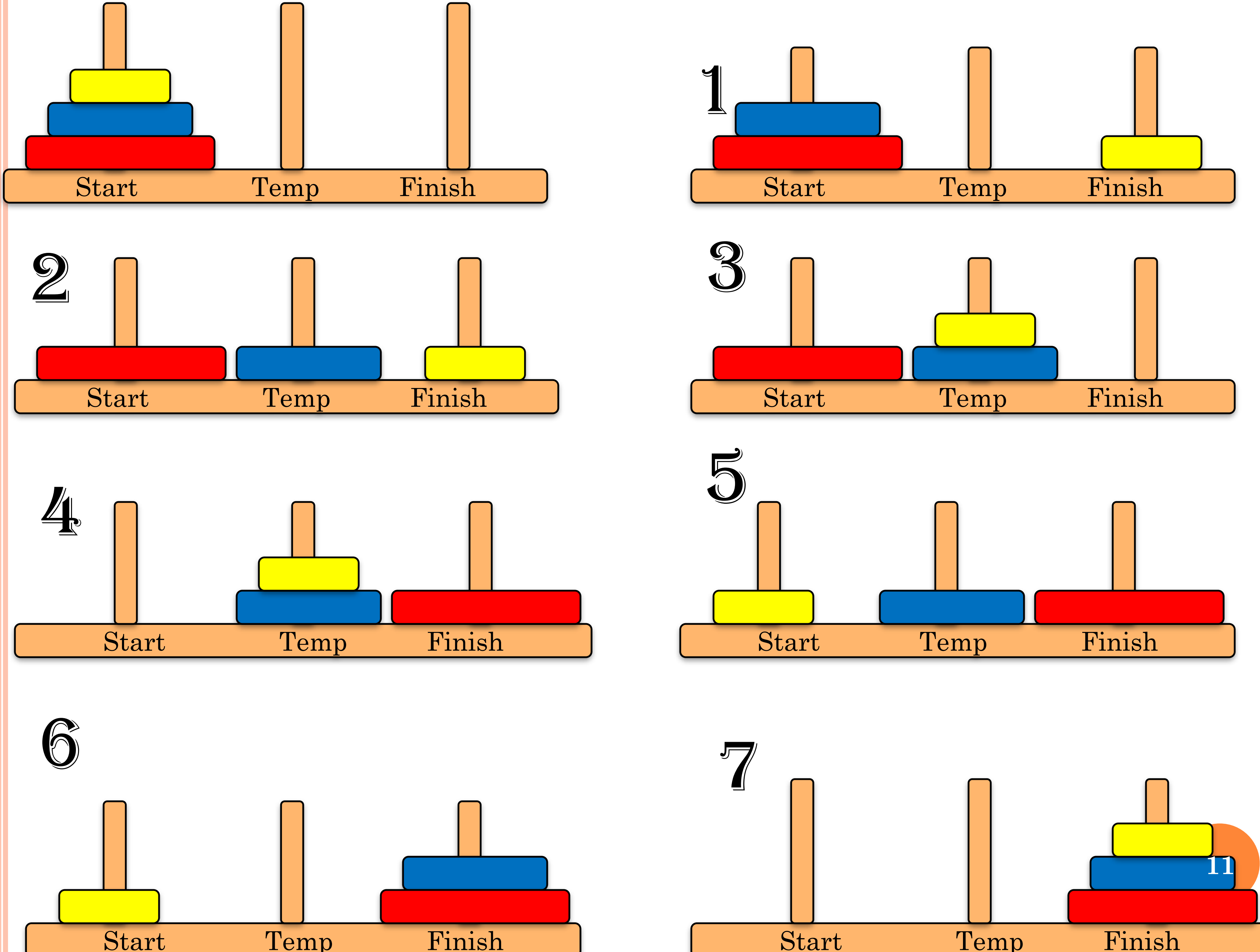


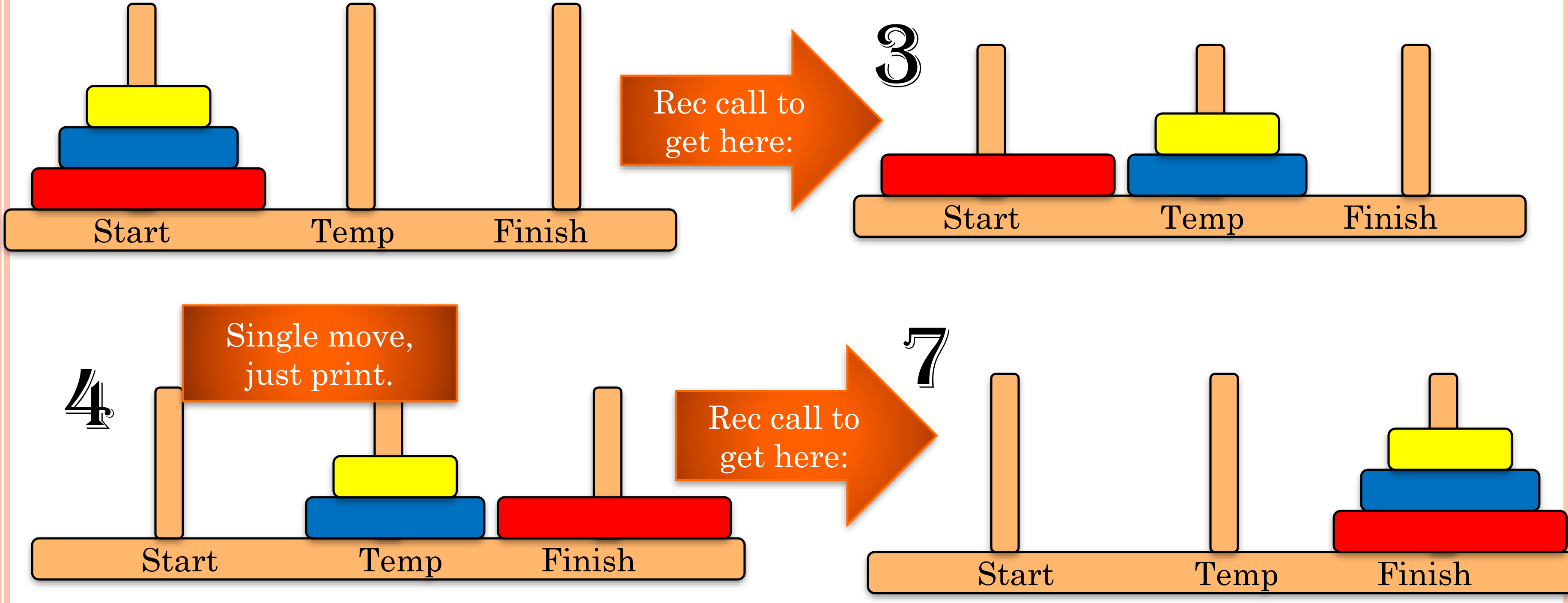




Start
Temp
Finish
1
Start
Temp
Finish
2
Start
Temp
Finish
3
Start
Temp
Finish
4
Start
Temp
Finish
5
Start
Temp
Finish
6
Start
Temp
Finish
7
Start
Temp
Finish

# Towers of Hanoi: Number of Steps

- Number of Steps:
  - 3 disks required 7 steps
  - 4 disks would requre 15 steps
  - We get n disks would require $\mathbf{2^n - 1}$ steps
    - HUGE number


Start
Temp
Finish
1
Start
Temp
Finish
2
Start
Temp
Finish
3
Start
Temp
Finish
4
Start
Temp
Finish
5
Start
Temp
Finish
6
Start
Temp
Finish
7
Start
Temp
Finish




```
void doHanoi(int n, char start, char finish, char temp) {
    if (n==1) {
            printf("Move Disk from %c to %c\n", start, finish);
    }
    else {
            doHanoi(n-1, start, temp, finish);
            printf("Move Disk from %c to %c\n, start finish);
            doHanoi(n-1, temp, finish, start);
    }
}
```

# Dynamic Programming

*An efficient way to implement some divide and conquer algorithms*

## Chapter Outcomes

- Understand the basic idea of dynamic programming
- Able to apply dynamic programming to solve different optimization problem
- **Strengths** and **weaknesses** of dynamic programming strategy

# Optimization Problems

- If a problem has only ***one correct solution***, then optimization is not required
- For example, there is only one sorted sequence containing a given set of numbers.
- **Optimization problems** have ***many solutions***.
- We want to compute an ***optimal solution*** ***e. g.*** with minimal cost and maximal gain.
- Dynamic programming is very effective technique.
- Development of dynamic programming algorithms can be broken into a sequence steps as in the next.

# Why Dynamic Programming over DAC?

- Dynamic programming, like divide and conquer method, solves problems by combining the solutions to sub-problems.
- Divide and conquer algorithms:
  - Partition the problem into independent or **disjoint** sub-problems
  - Solve the sub-problem recursively
  - Combine their solutions to solve the original problem
- In contrast, dynamic programming is applicable when the sub-problems are not independent , when the sub-problems overlap—that is, when sub-problems share sub-subproblems..
- Dynamic programming is typically applied to optimization problems.

# Dynamic programming (DP)

- It follows **principle of optimality** developed by Richard Bellman.
- It solves each sub-subproblem just once (just the first time) and then saves its answer in a table.
- And at any subsequent time if it needs to solves the same sub-subproblem – just use it from the table.
  - this simple idea can sometimes transform **exponential-time** algorithms into **polynomial-time** algorithms.
  - Otherwise it **will be normal brute force** technique.
- So, we can call DP a **smart/clever Brute force** technique.

# DYNAMIC PROGRAMMING (DP)

- Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.
  - Either maximize or minimize something

- Dynamic programming is effective when a given subproblem **may arise from more than one partial set** of choices;

- So, DP can be think of as
  - **Overlapped** subproblems that can be reused
  - **Exhaustive** search but in a **clever** way
    - As it will consider all possibilities in come to a solution not just one greedy choice.

# Elements of Dynamic Programming (DP)

DP is used to solve problems with the following characteristics:

- Simple subproblems
  - We should be able to break the original problem to **smaller subproblems** that have the **same** structure
- Optimal substructure of the problems
  - The **optimal solution** to the problem contains within **optimal solutions to** its **subproblems**.
- Overlapping sub-problems
  - there exist some places where we solve the **same subproblem** more than **once**.

# Steps to Designing a Dynamic Programming Algorithm

1. Characterize optimal substructure
2. Recursively define the value of an optimal solution
3. Compute the value bottom up
4. (if needed) Construct an optimal solution

# Steps to Designing a Dynamic Programming Algorithm

1. Characterize the structure of an optimal solution.
   - Define subproblem
   - Guess part of the solution
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
   - **Memoize** or **bottom-up** fashion
4. Construct an optimal solution from computed information.

# Steps to Designing a Dynamic Programming Algorithm

- Steps 1–3 form the basis of a dynamic-programming solution to a problem.
- If we need only the value of an optimal solution, and not the solution itself, then we can omit step 4.
  - When we do perform step 4, we sometimes maintain additional information during step 3 so that we can easily construct an optimal solution.

# Tabulation vs Memoization

- There are two different ways to store the values so that the values of a sub-problem can be reused. Here, will discuss two patterns of solving dynamic programming (DP) problems:

- **Tabulation:** Bottom Up
- **Memoization:** Top Down

## Tabulation vs Memoization

- Tabulation Method – Bottom Up Dynamic Programming

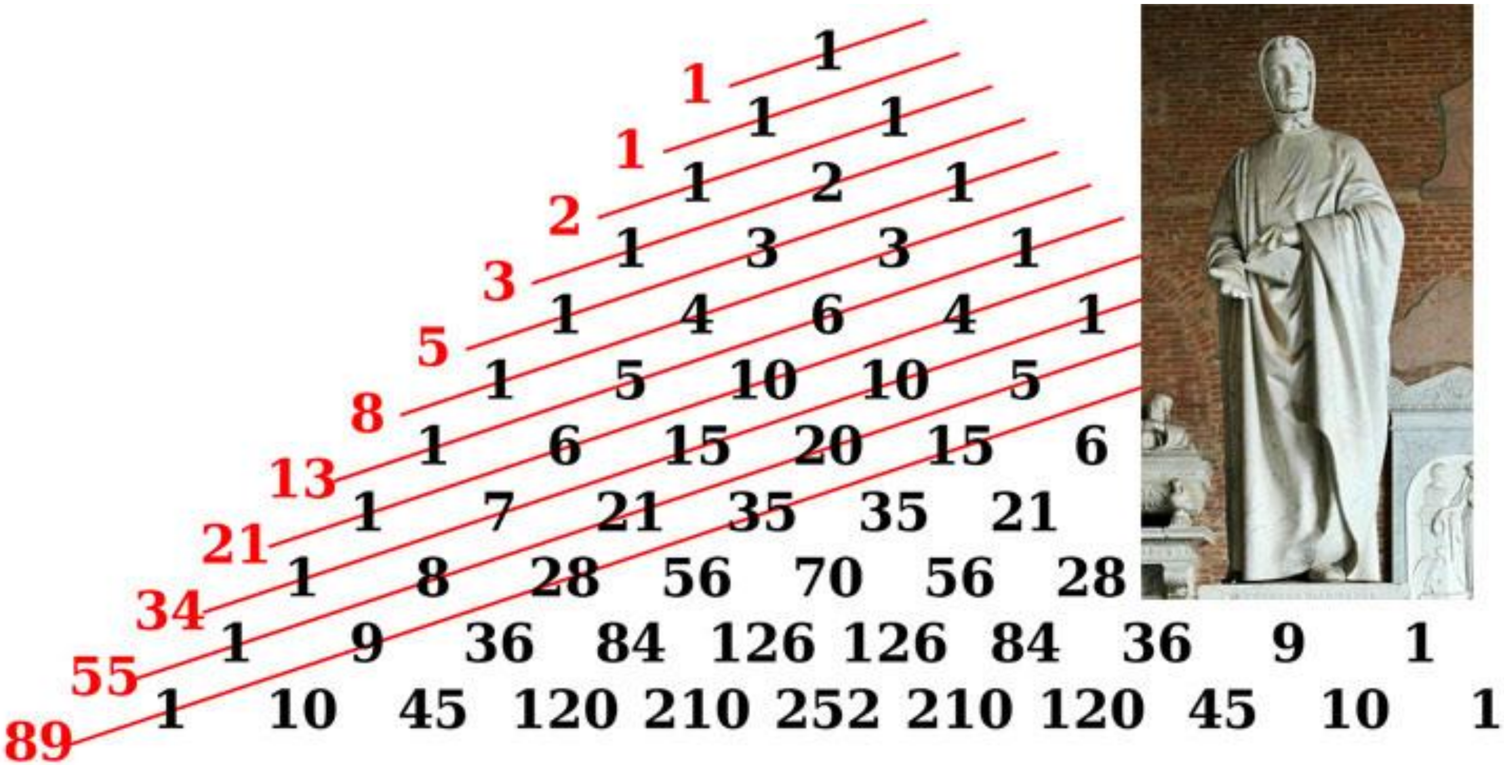
```
int dp[0] = 1;
for (int i = 1; i< =n; i++)
{
    dp[i] = dp[i-1] * i;
}
```

## Tabulation vs Memoization

- Memoization Method – Top-Down Dynamic Programming

```
// return fact x!
int solve(int x)
{
    if (x==0)
        return 1;
    if (dp[x]!=-1)
        return dp[x];
    return (dp[x] = x * solve(x-1));
}
```

# Fibonacci Number



# LET'S THINK ABOUT FIBONACCI NUMBER

In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

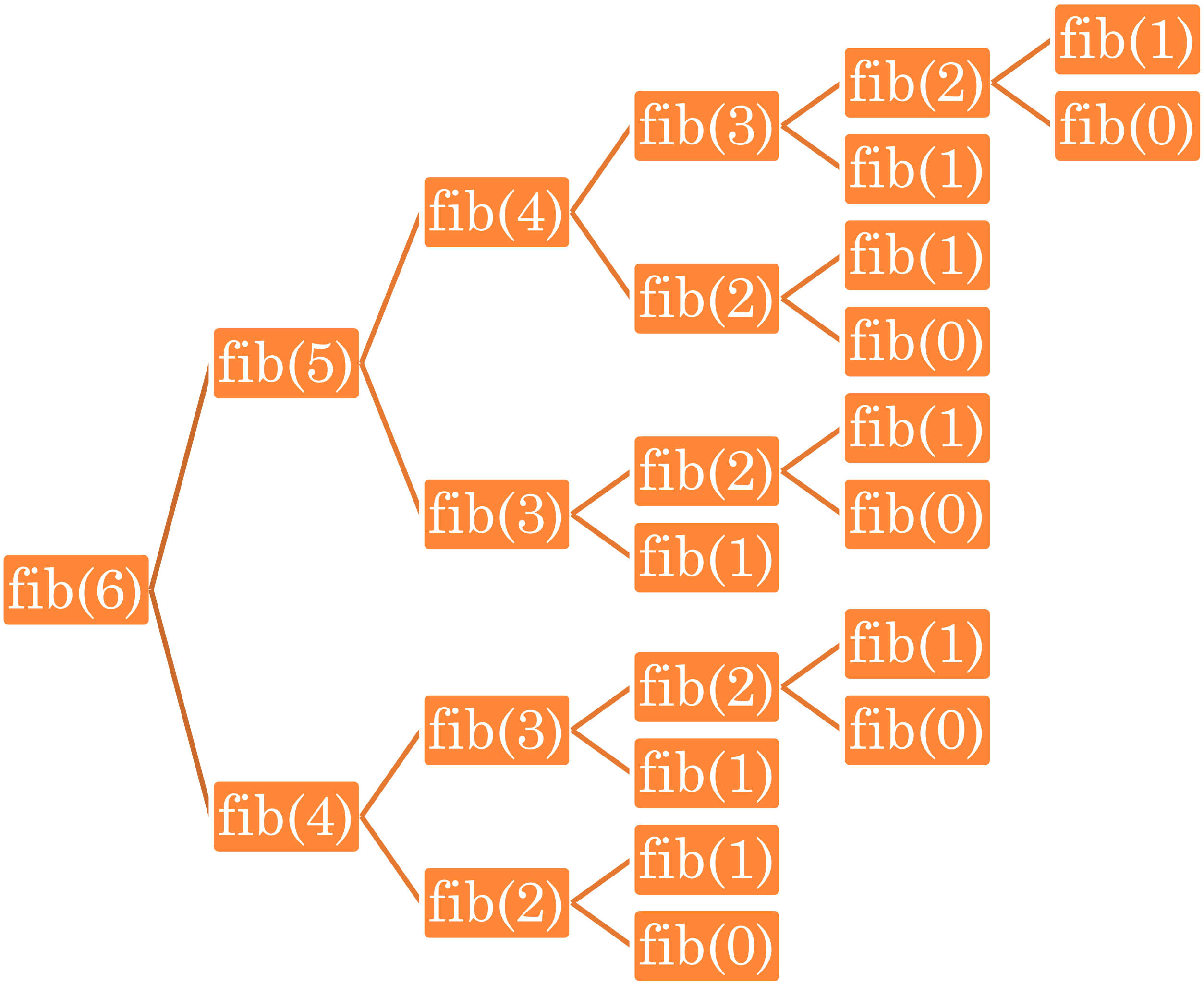
| n | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| F(n) | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 |

Pseudo code for the recursive algorithm:

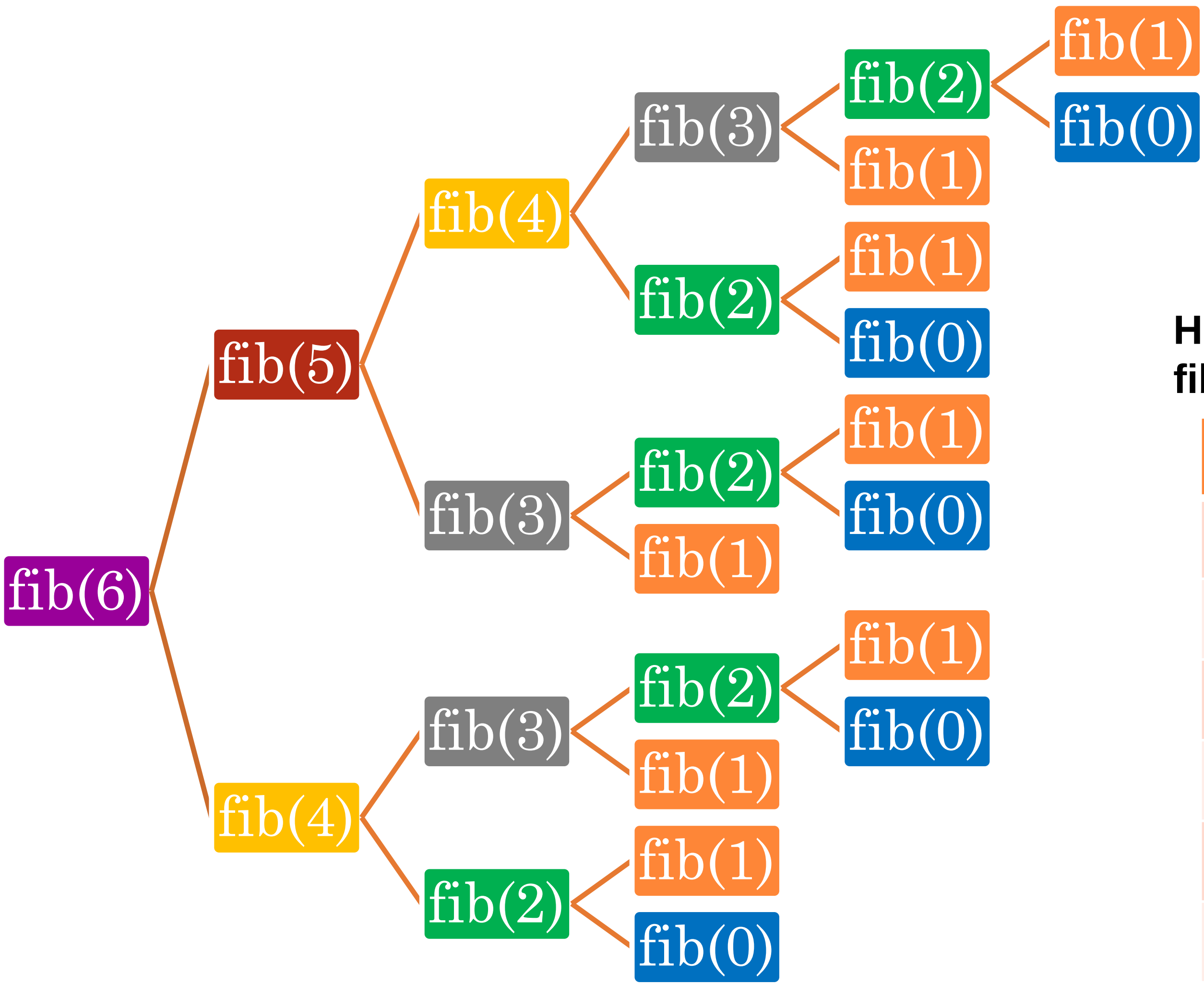
```
Procedure F(n)
    if n==0 or n==1 then
        return 1
    else
        return F(n-1) + F(n-2)
```

- Time Complexity: $\Theta(2^n)$
- Is it a good algorithm?
- Is there any way to improve?

# Fibonacci number – recursion tree for n=6



# Fibonacci number – recursion tree for n=6




**How many times each fib(n) is called?**

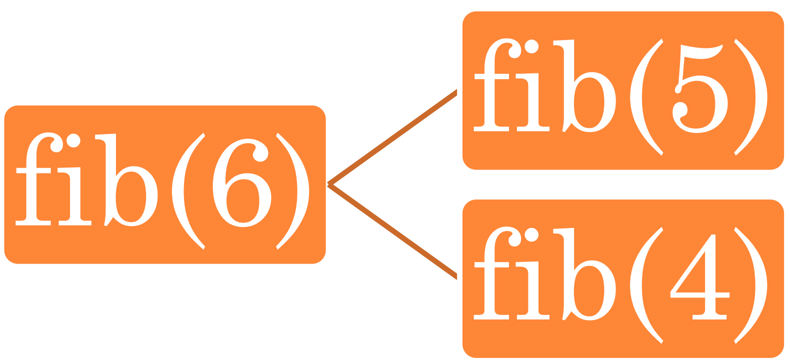
| Function | Count |
|---|---|
| fib(5) | 1 |
| fib(4) | 2 |
| fib(3) | 3 |
| fib(2) | 5 |
| fib(1) | 7 |
| fib(0) | 5 |

# Improvement – memoization (remembering)

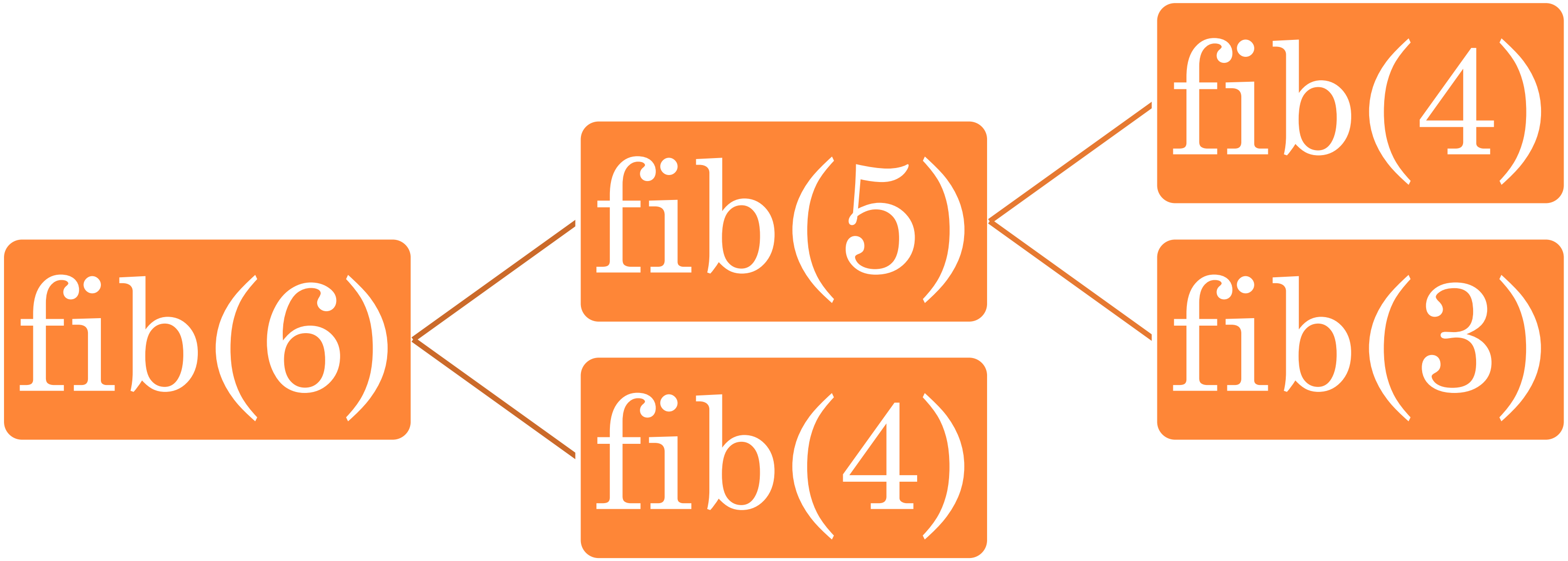
DP : using memorization (Top down approach)

- Calculate once
- Store it
- And reuse it

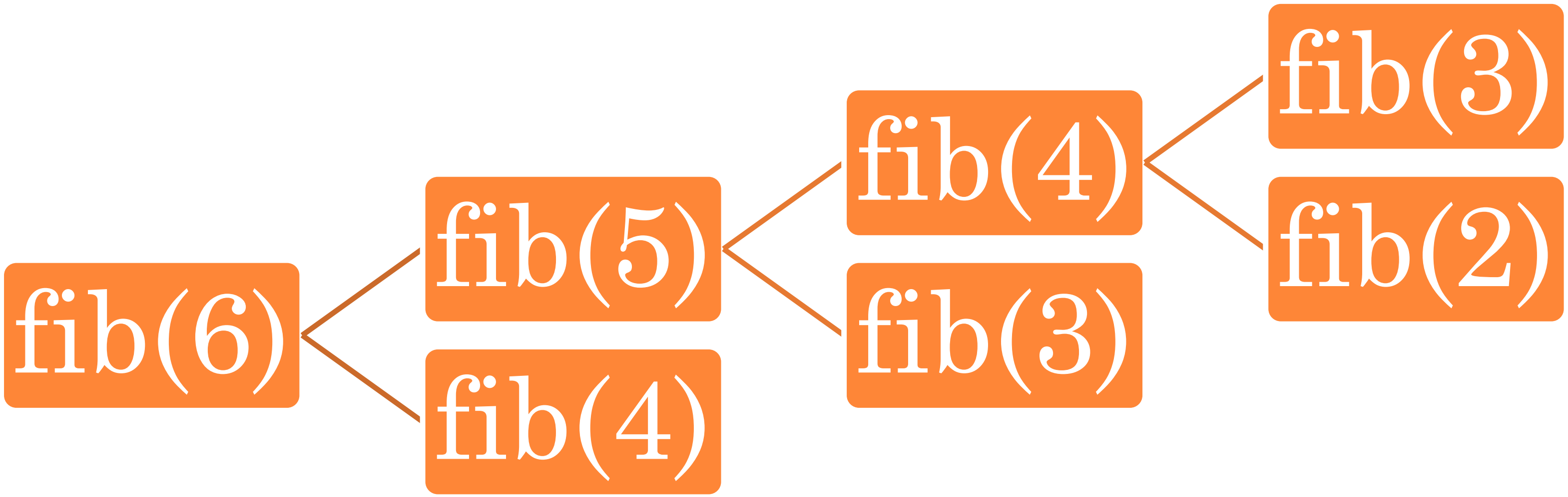
# Fibonacci number – divide step



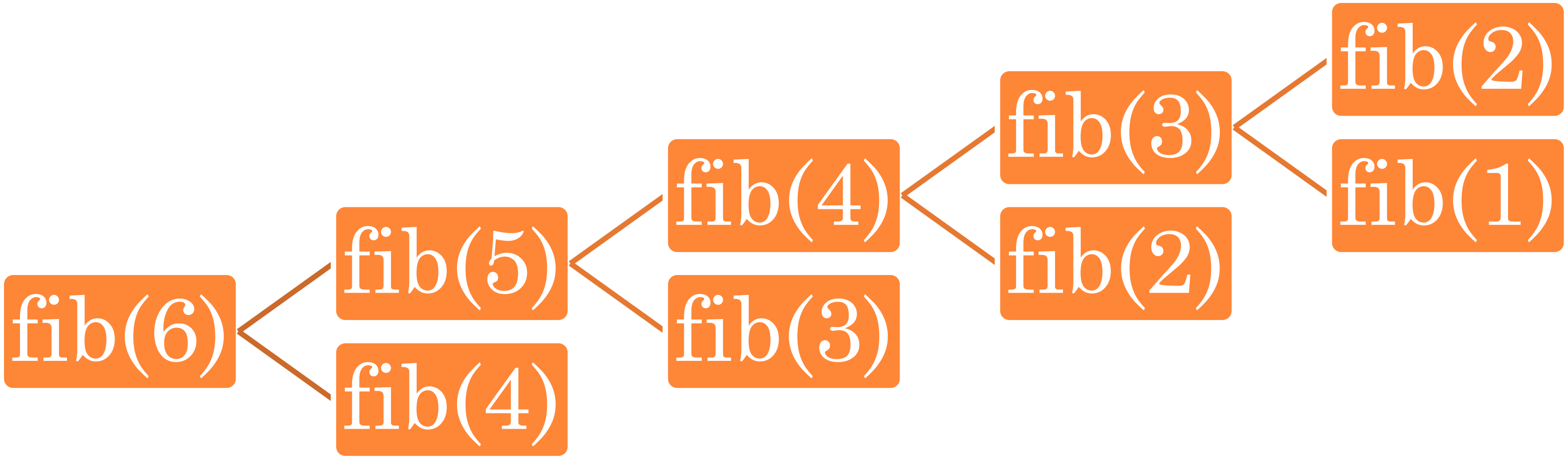
# Fibonacci number – divide step Cont..



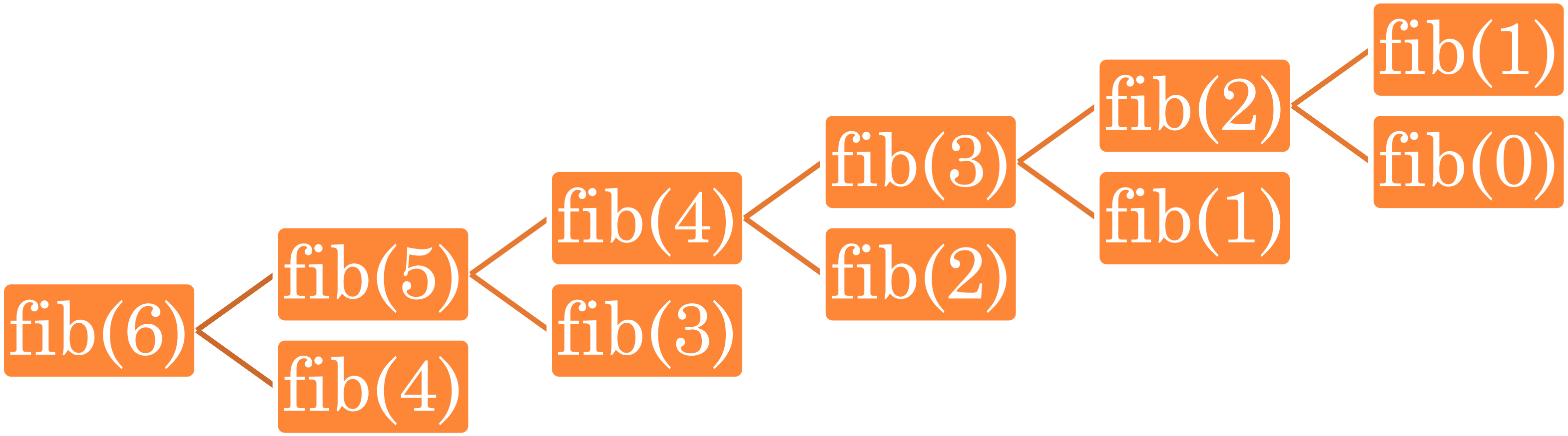
# Fibonacci number – divide step Cont..



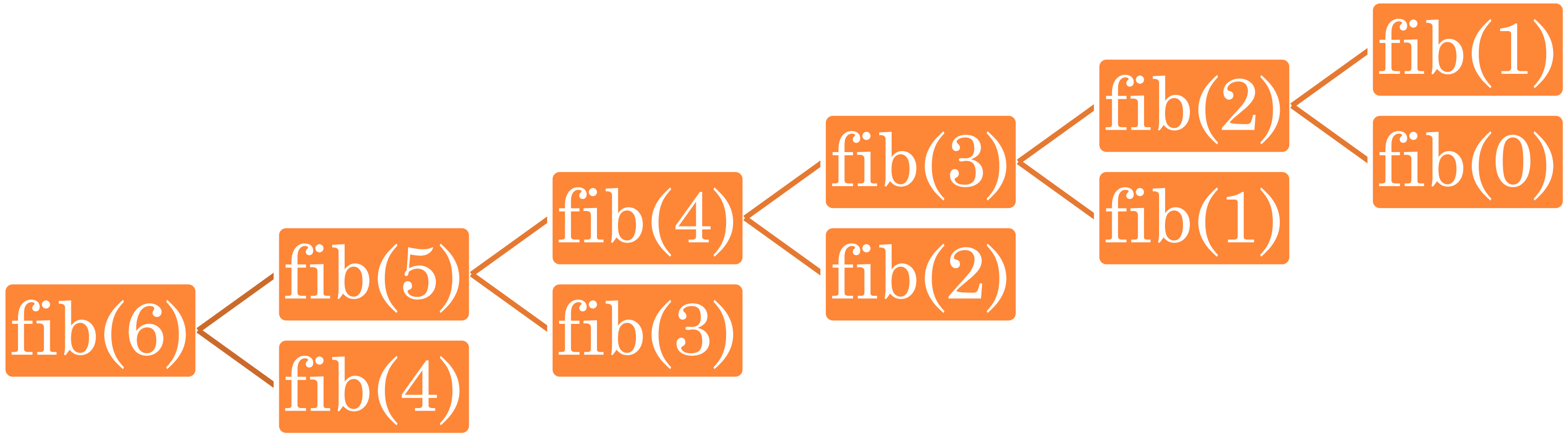
# Fibonacci number – divide step Cont..



# Fibonacci number – divide step Cont..

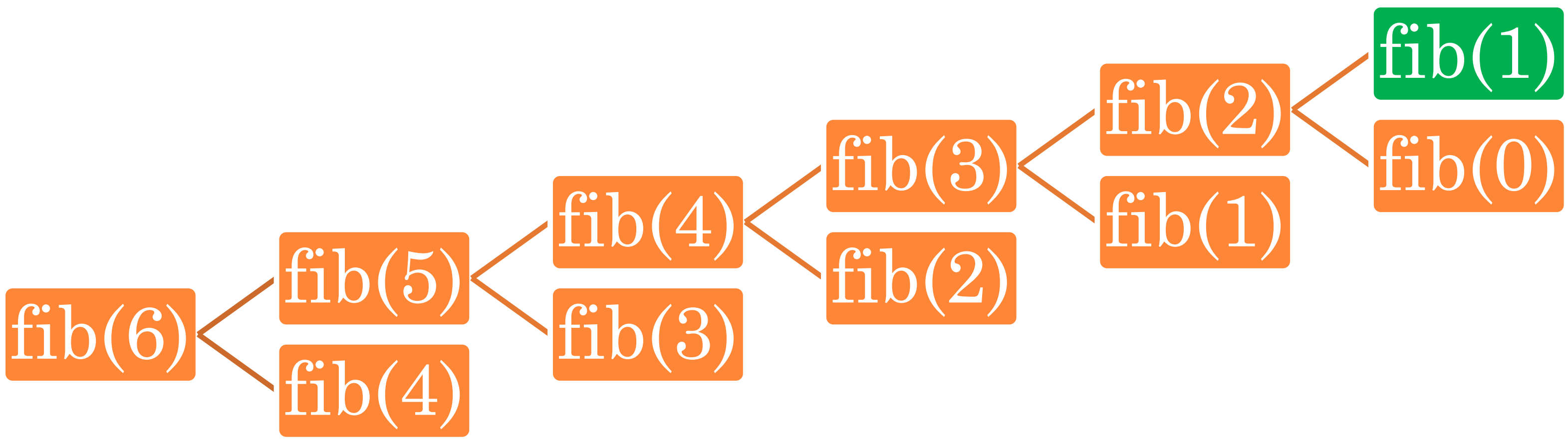


# FIBONACCI NUMBER – DIVIDE STEP CONT..




| Function | value |
|---|---|
| fib(6) | |
| fib(5) | |
| fib(4) | |
| fib(3) | |
| fib(2) | |
| fib(1) | |
| fib(0) | |

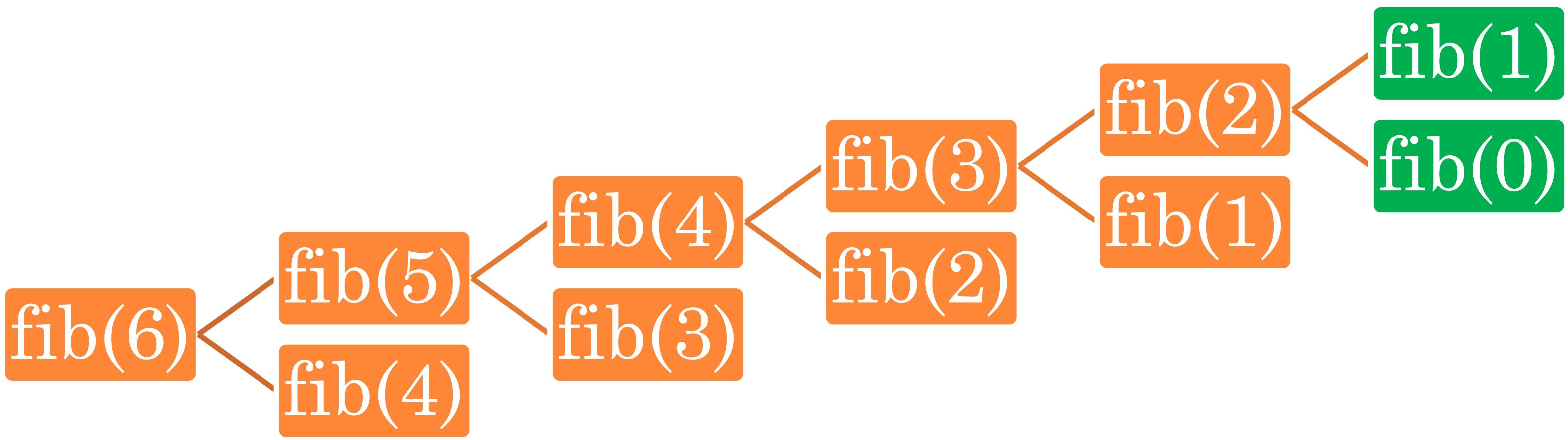
# Fibonacci number – conquer step




| Function | value |
|---|---|
| fib(6) | |
| fib(5) | |
| fib(4) | |
| fib(3) | |
| fib(2) | |
| fib(1) | 1 |
| fib(0) | |

fib(n) Indicates calculated and saved to table

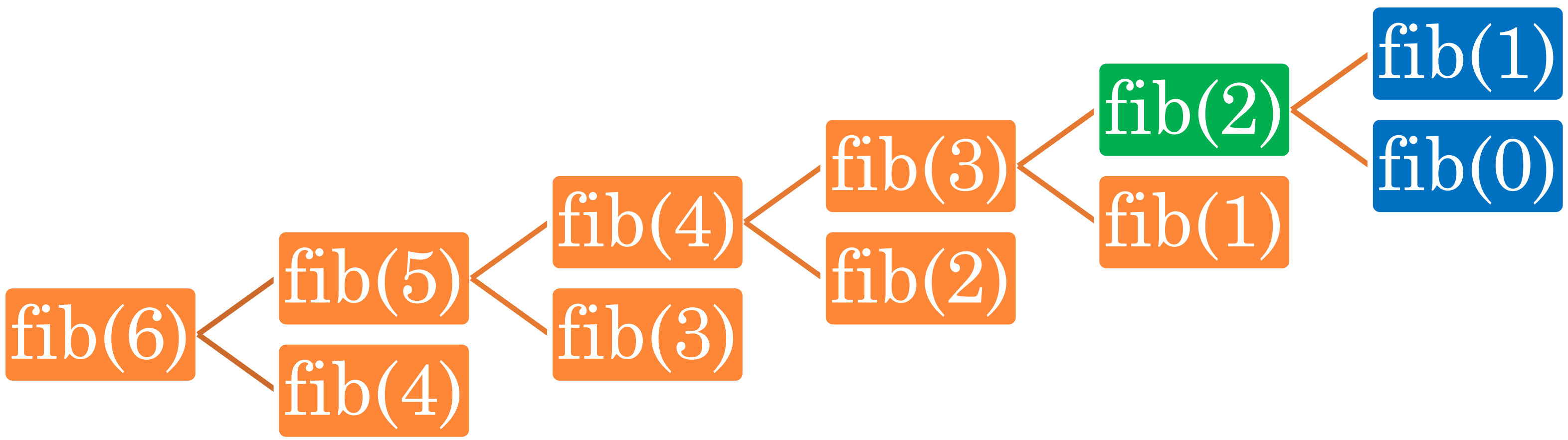
# CONQUER STEP CONT..




fib(n) Indicates calculated and saved to table

| Function | value |
|---|---|
| fib(6) | |
| fib(5) | |
| fib(4) | |
| fib(3) | |
| fib(2) | |
| fib(1) | 1 |
| fib(0) | 0 |

# Combine & conquer step Cont..

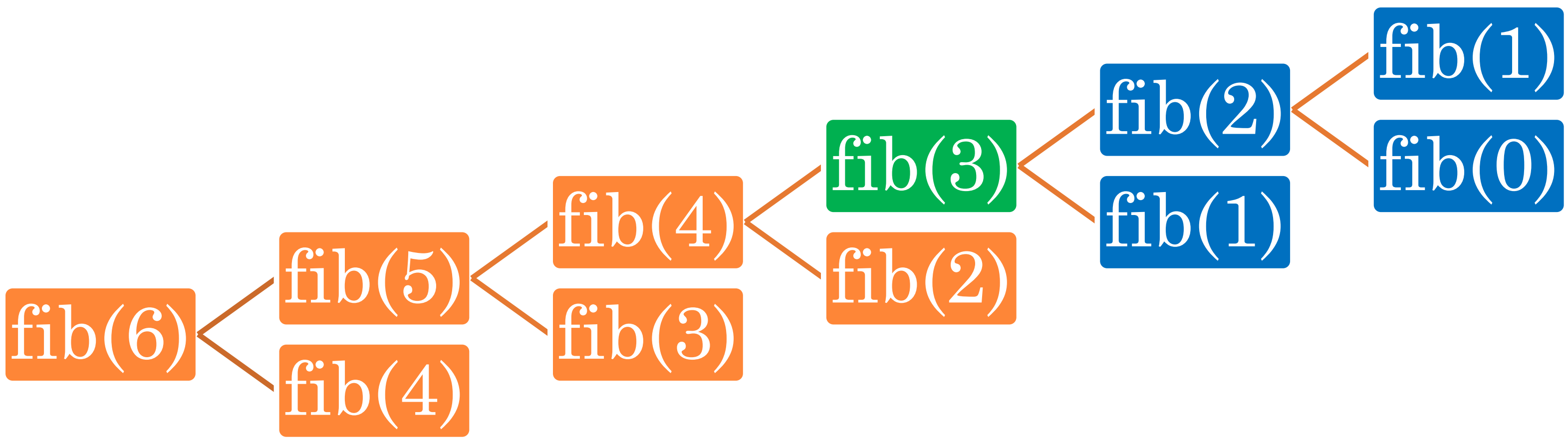



| Function | value |
|---|---|
| fib(6) | |
| fib(5) | |
| fib(4) | |
| fib(3) | |
| fib(2) | 1 |
| fib(1) | 1 |
| fib(0) | 0 |

fib(n) Indicates calculated and saved to table

fib(n) Indicates using the saved data

# Combine & conquer step Cont..

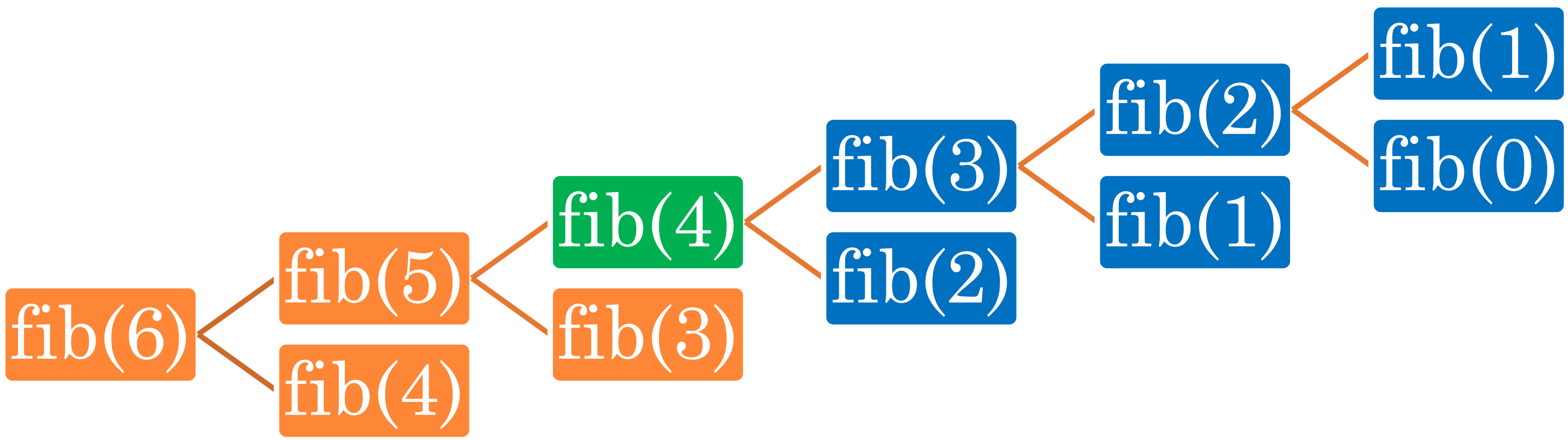



| Function | value |
|---|---|
| fib(6) | |
| fib(5) | |
| fib(4) | |
| fib(3) | 2 |
| fib(2) | 1 |
| fib(1) | 1 |
| fib(0) | 0 |

fib(n) Indicates calculated and saved to table

fib(n) Indicates using the saved data

# Combine & conquer step Cont..

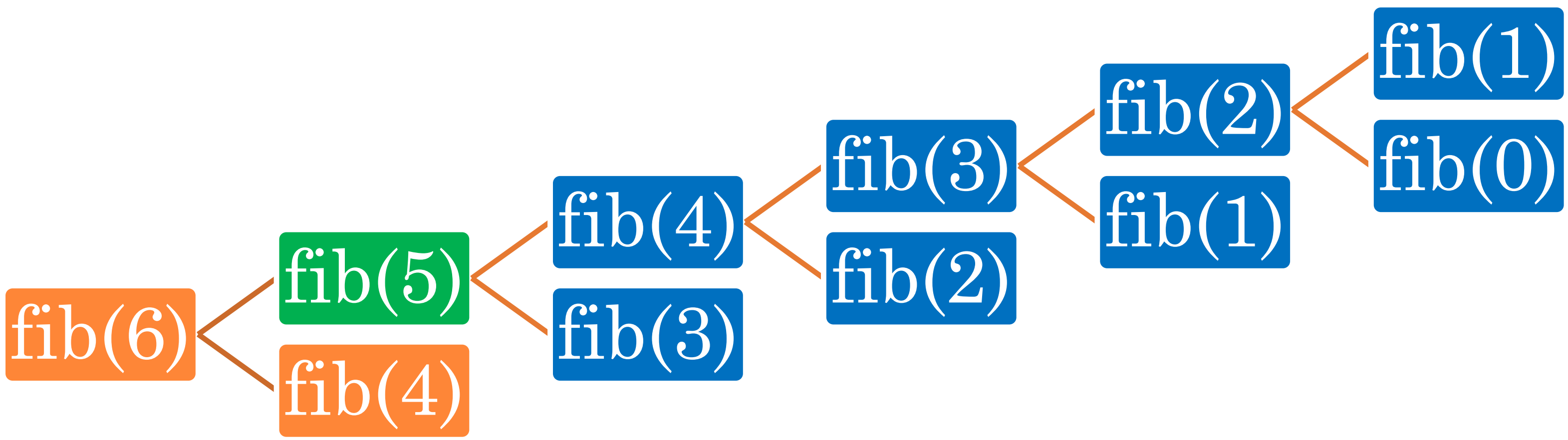



| Function | value |
|---|---|
| fib(6) | |
| fib(5) | |
| fib(4) | 3 |
| fib(3) | 2 |
| fib(2) | 1 |
| fib(1) | 1 |
| fib(0) | 0 |

fib(n) Indicates calculated and saved to table

fib(n) Indicates using the saved data

# Combine & conquer step Cont..

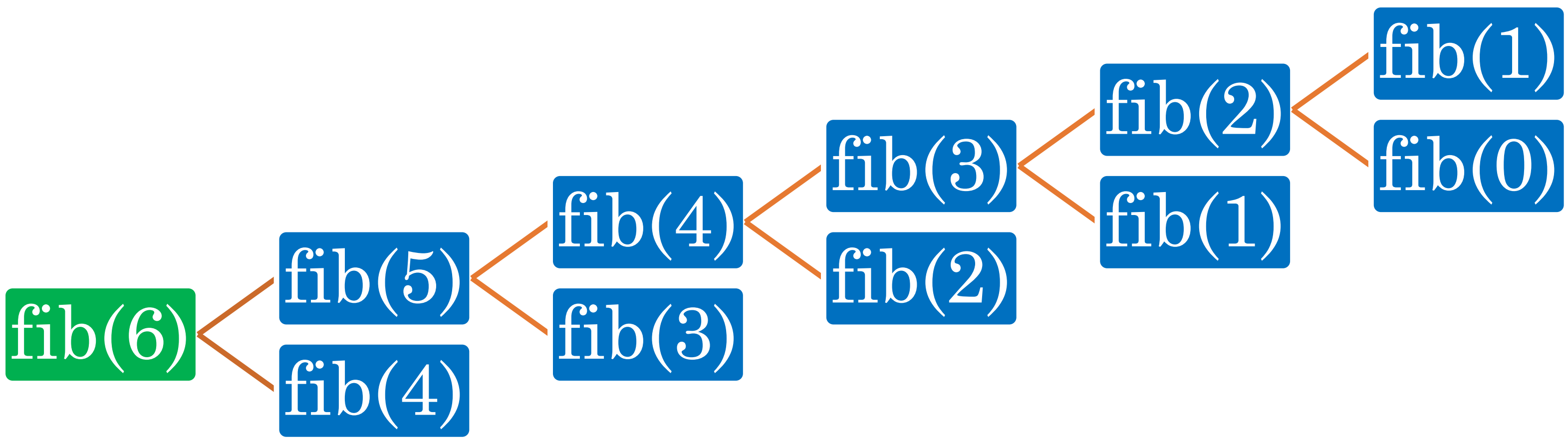



fib(n) Indicates calculated and saved to table

fib(n) Indicates using the saved data

| Function | value |
|---|---|
| fib(6) | |
| fib(5) | 5 |
| fib(4) | 3 |
| fib(3) | 2 |
| fib(2) | 1 |
| fib(1) | 1 |
| fib(0) | 0 |

# Combine & conquer step Cont..




| Function | value |
|---|---|
| fib(6) | 8 |
| fib(5) | 5 |
| fib(4) | 3 |
| fib(3) | 2 |
| fib(2) | 1 |
| fib(1) | 1 |
| fib(0) | 0 |

fib(n) Indicates calculated and saved to table

fib(n) Indicates using the saved data

# IDEA FOR IMPROVEMENT

Memorization:

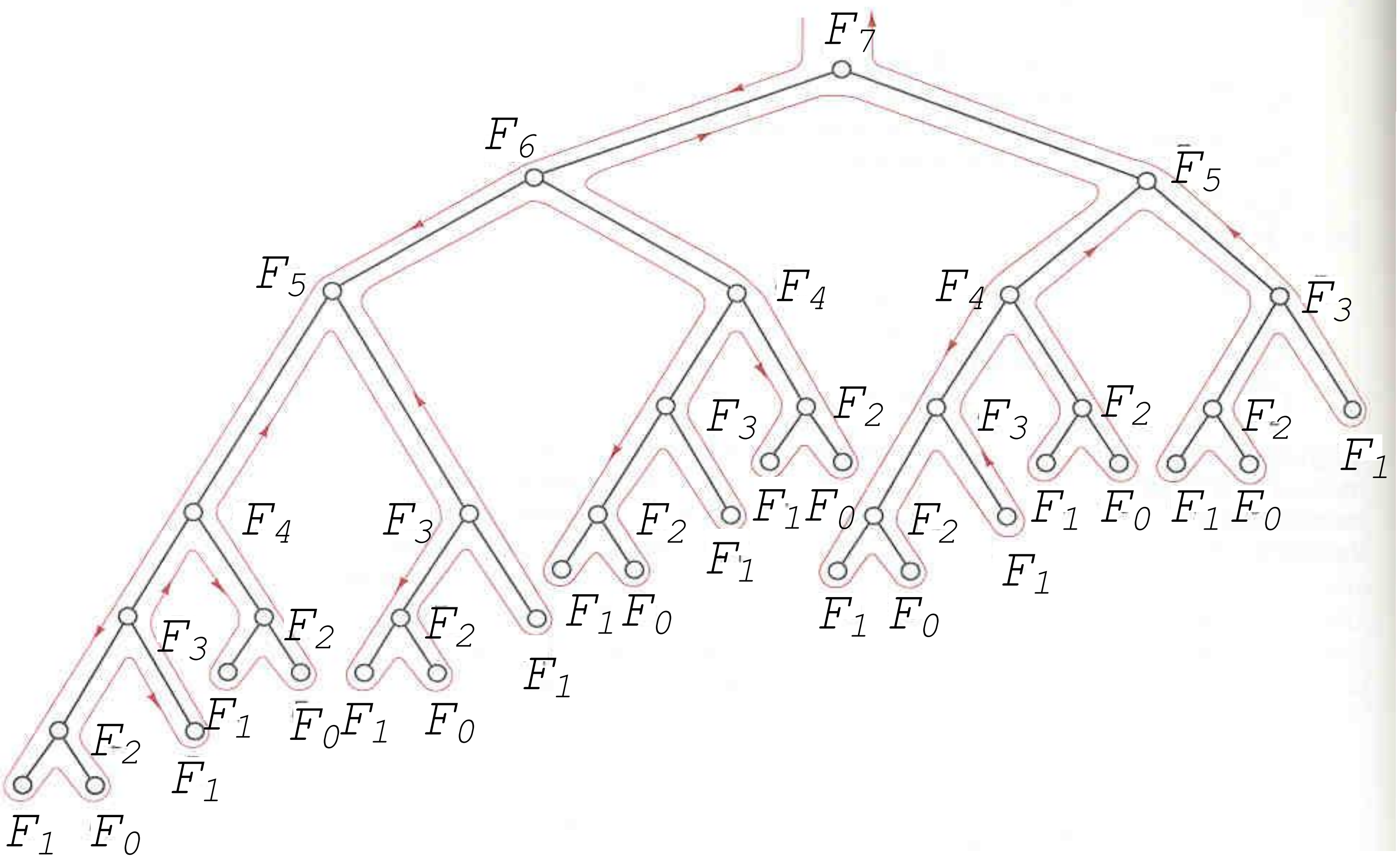
- Store F(i) somewhere after we have computed its value
- Afterward, we don't need to re-compute F(i); we can retrieve its value from our memory.

```
Procedure F(n)
  if (v[n] < 0) then
    v[n] =  F(n-1)+F(n-2)
  return v[n]
```

```
Main
  set v[0] = v[1] = 1
  for i = 2 to n do
    v[i] = -1
  output F(n)
```

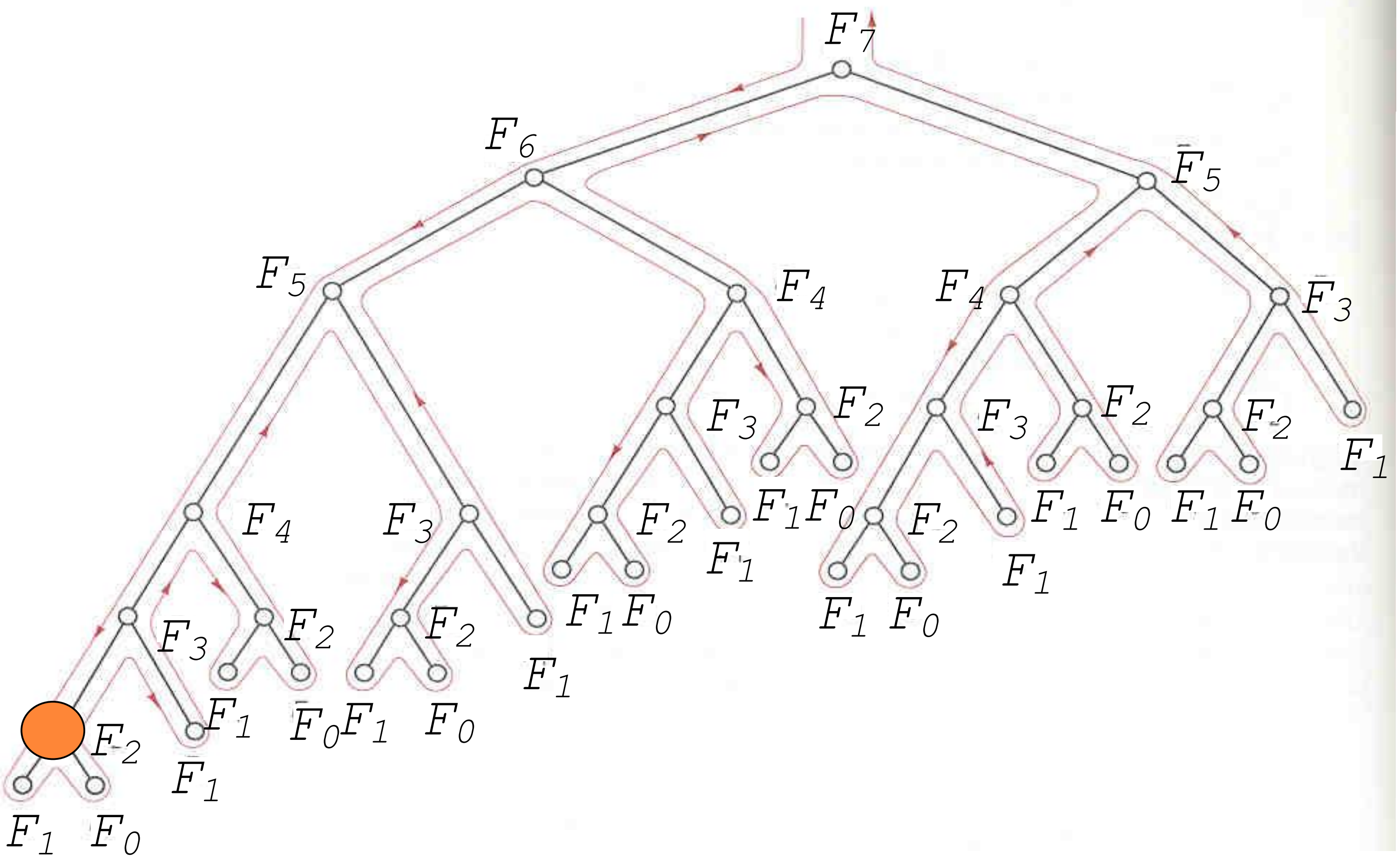
[ ] refers to array
( ) is parameter for calling a procedure

# Look at the execution of F(7)



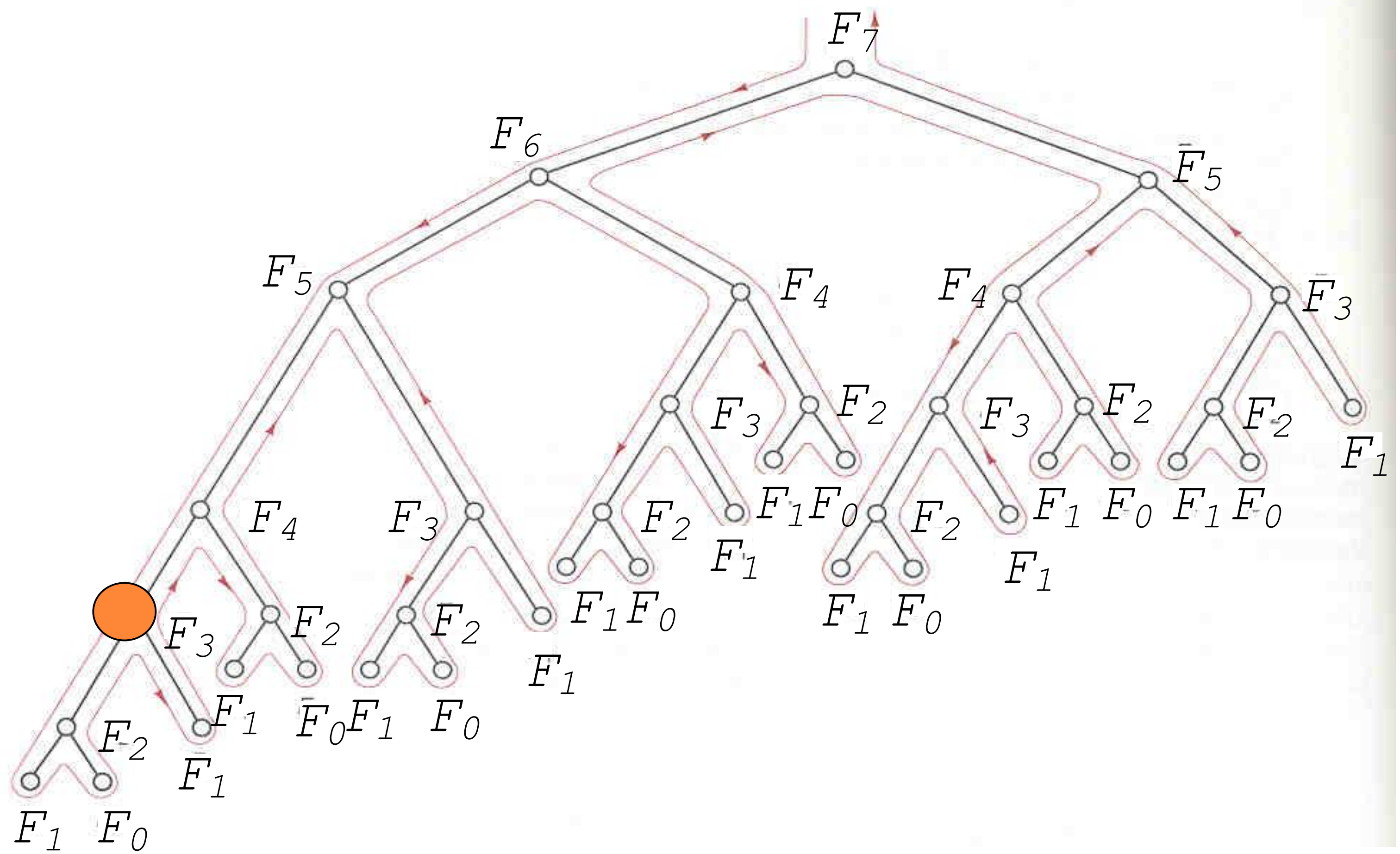

| | |
|---|---|
| v[0] | 1 |
| v[1] | 1 |
| v[2] | -1 |
| v[3] | -1 |
| v[4] | -1 |
| v[5] | -1 |
| v[6] | -1 |
| v[7] | -1 |

# Look at the execution of F(7)



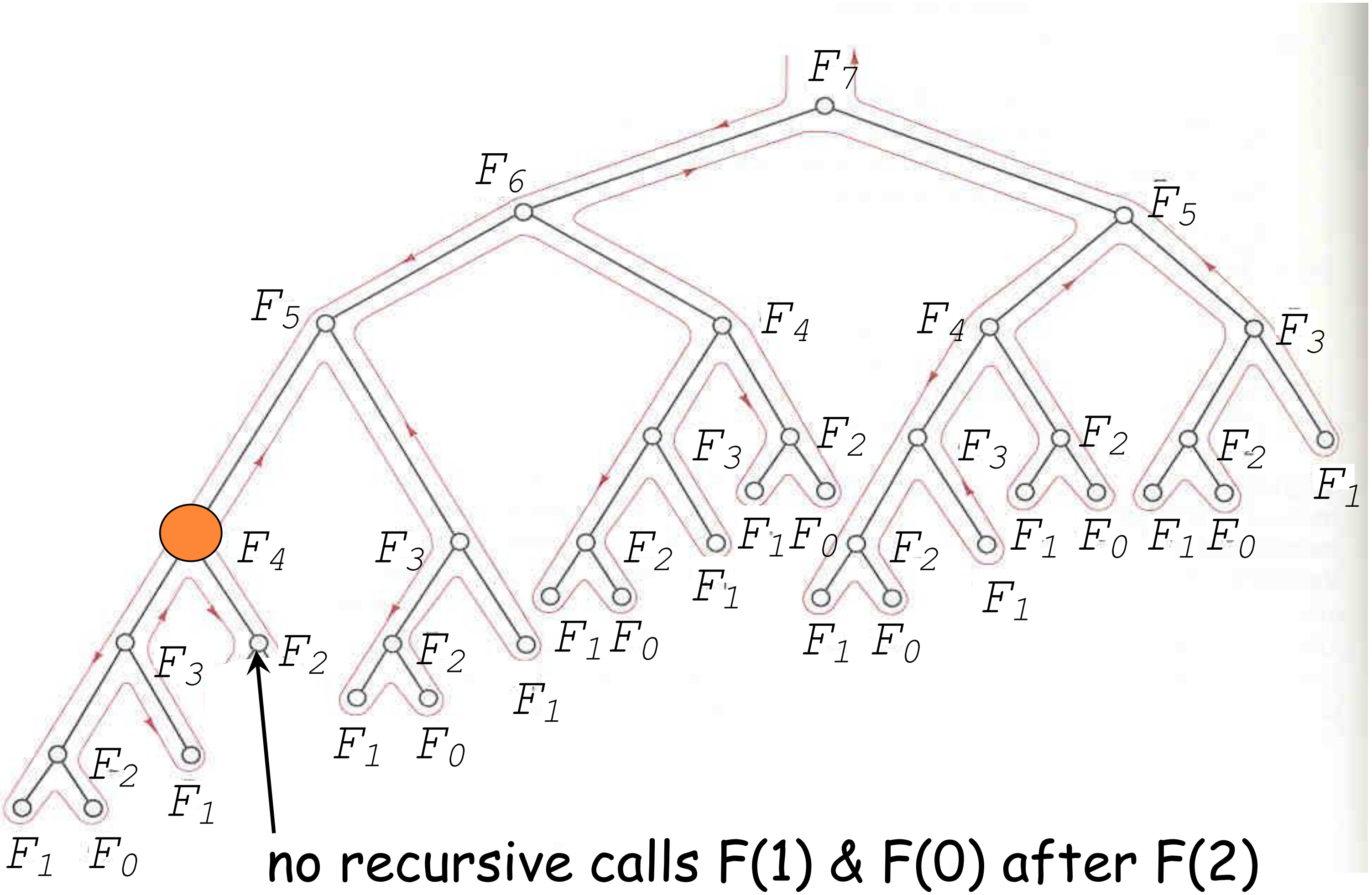

| | |
|---|---|
| v[0] | 1 |
| v[1] | 1 |
| v[2] | **2** |
| v[3] | -1 |
| v[4] | -1 |
| v[5] | -1 |
| v[6] | -1 |
| v[7] | -1 |

# Look at the execution of F(7)



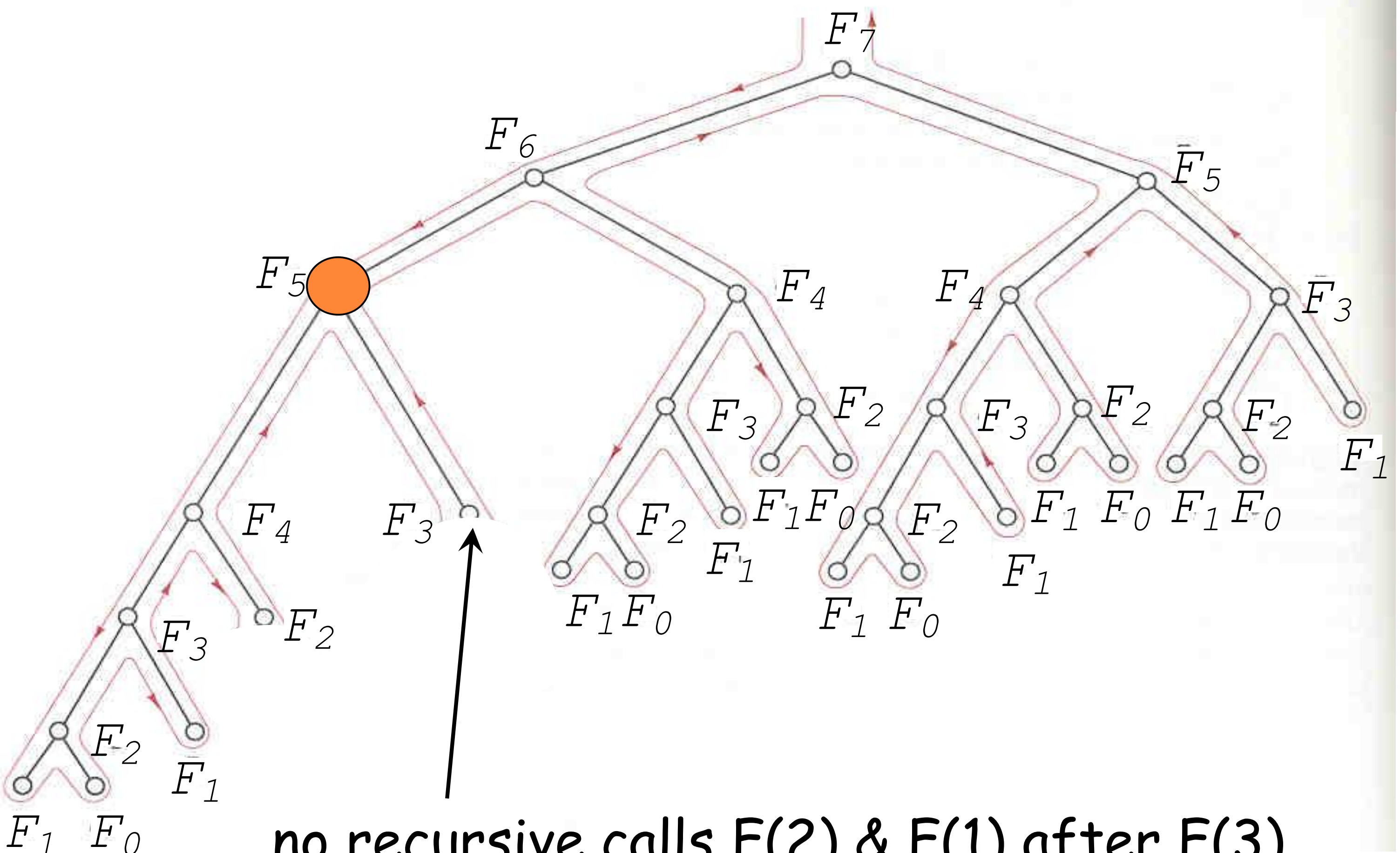

| | |
|---|---|
| v[0] | 1 |
| v[1] | 1 |
| v[2] | 2 |
| v[3] | **3** |
| v[4] | -1 |
| v[5] | -1 |
| v[6] | -1 |
| v[7] | -1 |

# Look at the execution of F(7)




| | |
|---|---|
| v[0] | 1 |
| v[1] | 1 |
| v[2] | 2 |
| v[3] | 3 |
| v[4] | **5** |
| v[5] | -1 |
| v[6] | -1 |
| v[7] | -1 |

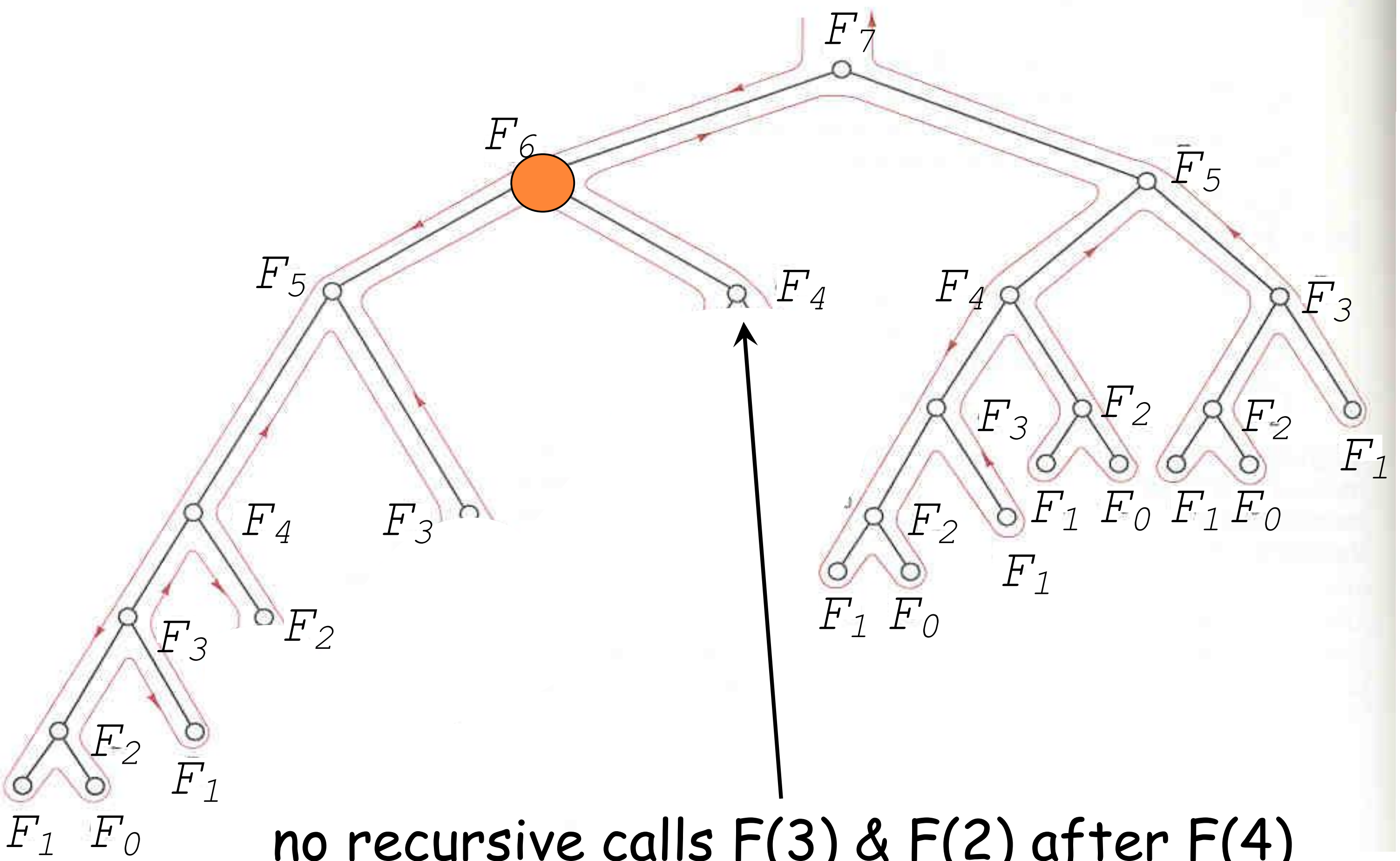
# Look at the execution of F(7)




no recursive calls F(2) & F(1) after F(3)

| | |
|---|---|
| v[0] | 1 |
| v[1] | 1 |
| v[2] | 2 |
| v[3] | 3 |
| v[4] | 5 |
| v[5] | **8** |
| v[6] | -1 |
| v[7] | -1 |

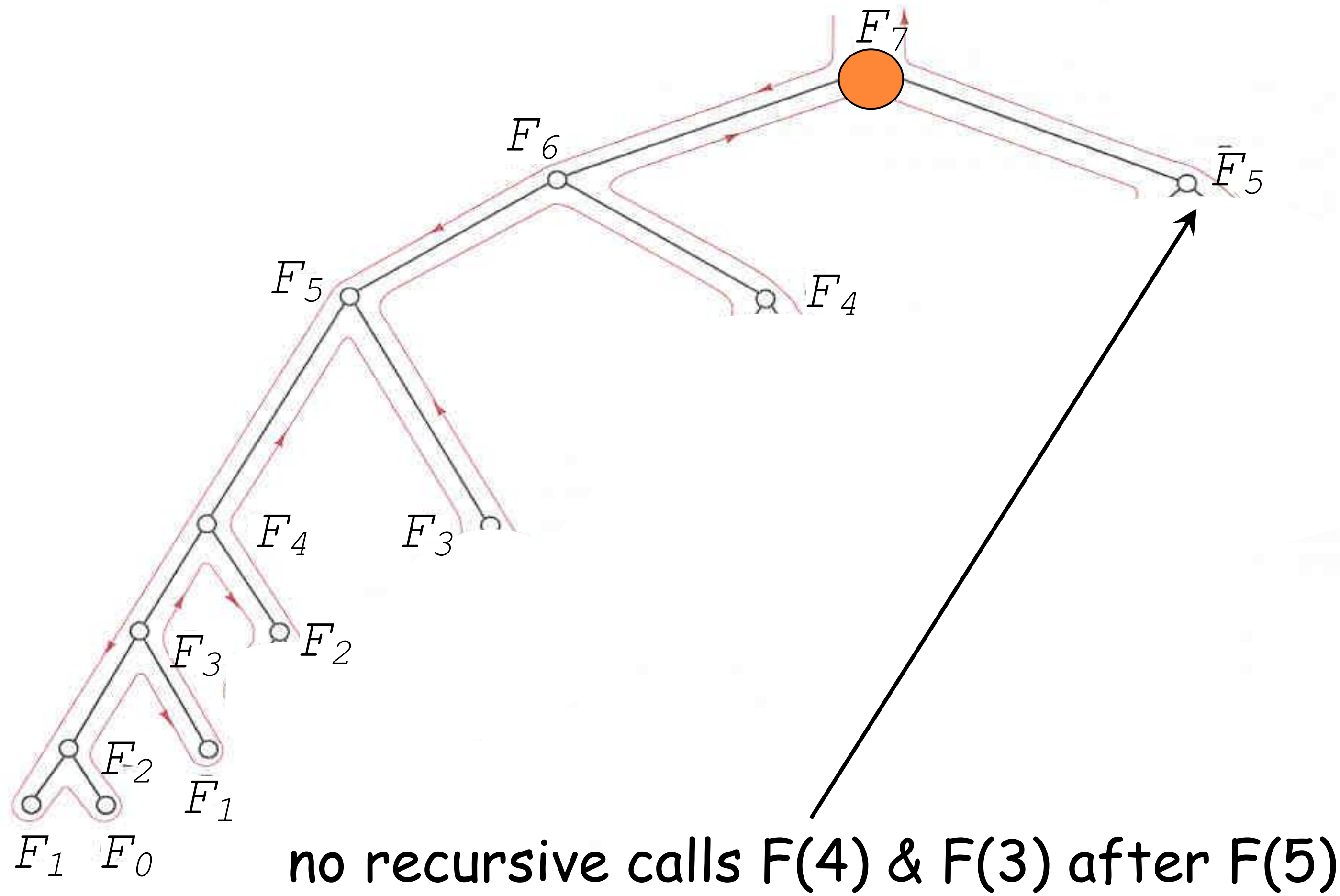
# LOOK AT THE EXECUTION OF F(7)




no recursive calls F(3) & F(2) after F(4)

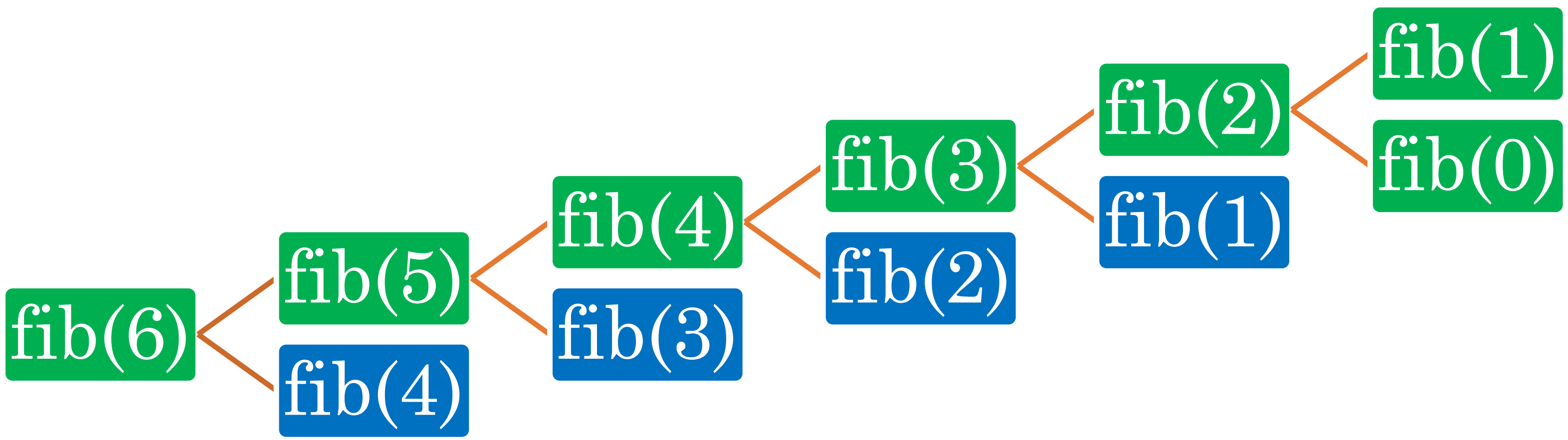
| | |
|---|---|
| v[0] | 1 |
| v[1] | 1 |
| v[2] | 2 |
| v[3] | 3 |
| v[4] | 5 |
| v[5] | 8 |
| v[6] | **13** |
| v[7] | -1 |

# LOOK AT THE EXECUTION OF F(7)




| | |
|---|---|
| v[0] | 1 |
| v[1] | 1 |
| v[2] | 2 |
| v[3] | 3 |
| v[4] | 5 |
| v[5] | 8 |
| v[6] | 13 |
| v[7] | **21** |

# Combine & conquer step Cont..




fib(n) Indicates the steps where we calculated.

- Time Complexity= $\Theta(n)$

# Recursive vs DP approach

Recursive version:

```
Procedure F(n)
    if n==0 or n==1 then
        return 1
    else
        return F(n-1) + F(n-2)
```

Too Slow!
exponential

Dynamic Programming version:

```
Procedure F(n)
    Set A[0] = A[1] = 1
    for i = 2 to n do
        A[i] = A[i-1] + A[i-2]
    return A[n]
```

Efficient!
Time complexity is $O(n)$

# When do we use memorization

- When a problem has following 2 properties:
  - **Optimal Substructure**: A problem depends on the solution of the sub-problems.
  - **Overlapping Sub-tructure**: Sub-problems are called several times.

# The Maximum Sub-Array

# MAXIMUM SUBARRAY

## Efficient solutions

Five solutions for this problem:-

1. Brute force approach I : Using 3 nested loops
2. Brute force approach II : Using 2 nested loops
3. Divide and Conquer approach : Similar to merge sort
4. Dynamic Programming Approach I : Using an auxiliary array
5. Dynamic Programming Approach II : Kadanes's Algorithm

# Maximum Subarray

| Solution Approach | Time Complexity | Space Complexity |
|---|---|---|
| Brute Force approach 1 | O(n^3) | O(1) |
| Brute Force approach 2 | O(n^2) | O(1) |
| Divide and Conquer Approach | O(nlogn) | O(logn) |
| Dynamic Programming using auxiliary array | O(n) | O(n) |
| Kadane Algorithm | O(n) | O(1) |

# Maximum Subarray– Kadane's Algorithm

## Kadane's Algorithm

Simple idea of the Kadane's algorithm is to look for all positive contiguous segments of the array (max_ending_here is used for this).

And keep track of maximum sum contiguous segment among all positive segments (max_so_far is used for this).

Each time we get a positive sum compare it with max_so_far and update max_so_far if it is greater than max_so_far

# Maximum Subarray– Kadane's Algorithm

```
Initialize:
    max_so_far = INT_MIN
    max_ending_here = 0

Loop for each element of the array
        (a) max_ending_here = max_ending_here + a[i]

        (b) if(max_so_far < max_ending_here)
                  max_so_far = max_ending_here

        (c) if(max_ending_here < 0)
                  max_ending_here = 0
return max_so_far
```

# Maximum Subarray– Kadane's Algorithm

```
int maxSubArraySum(int a[], int size)
{
   int max_so_far = INT_MIN, max_ending_here = 0;

   for (int i = 0; i < size; i++)
   {
      max_ending_here = max_ending_here + a[i];
      if (max_so_far < max_ending_here)
         max_so_far = max_ending_here;

      if (max_ending_here < 0)
         max_ending_here = 0;
   }
   return max_so_far;
}
```

Notice that each element has been visited only once.

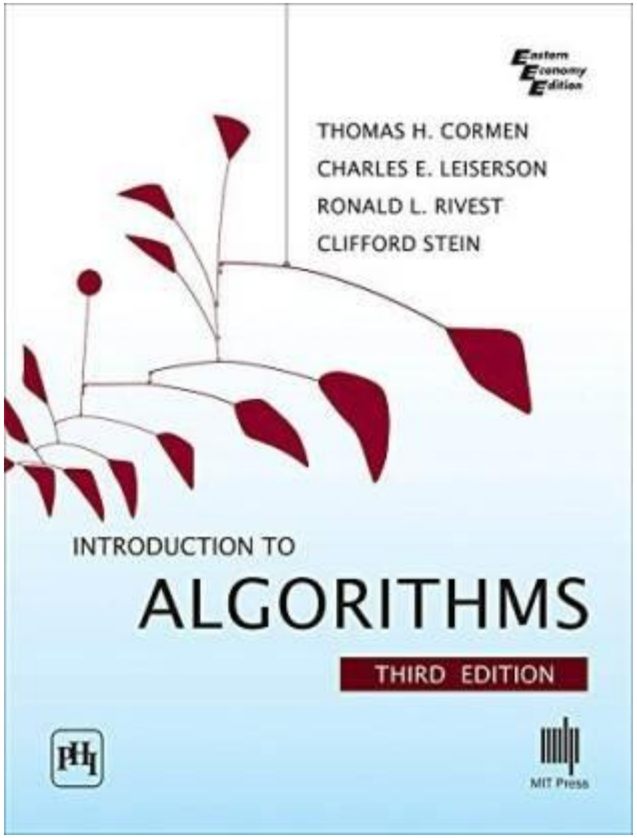
Time Complexity = O(n)

# Task to think

Find the longest subarray in a binary array with an equal number of 0s and 1s

# Reference

- Chapter 15 (15.1 and 15.3)
- **Introduction to Algorithms, 3rd Edition** Thomas H. Cormen




# Thanks to All