

Dynamic Programming I

SWE2016-44

Mathematical Background

Consider the following mathematical function:

$$f(n) = f(n - 1) + f(n - 2), \quad \text{where } n \geq 2$$

$$f(1) = 1 \quad \text{and} \quad f(0) = 0$$

Mathematical Background

Consider the following mathematical function:

$$f(n) = f(n - 1) + f(n - 2), \quad \text{where } n \geq 2$$

$$f(1) = 1 \quad \text{and} \quad f(0) = 0$$

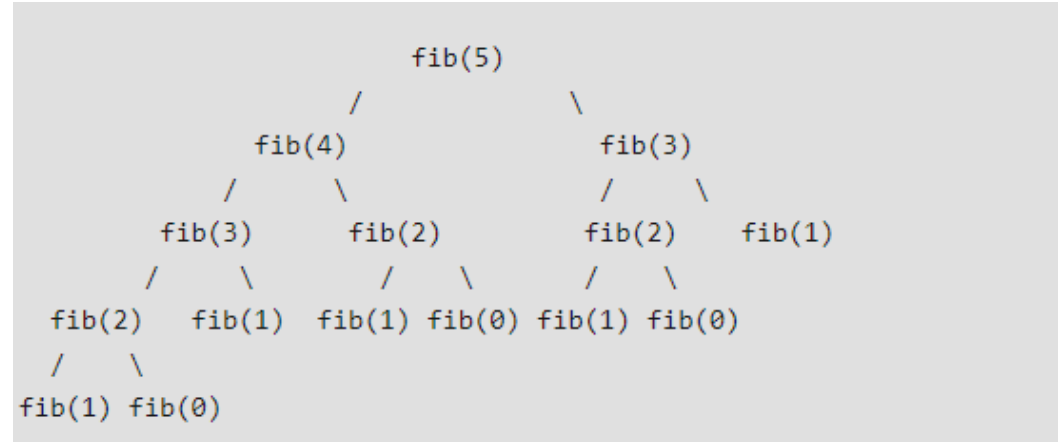
Fibonacci series: a series of numbers in which each number (Fibonacci number) is the sum of the two preceding numbers.

$$f(n) = \{0, 1, 1, 2, 3, 5, 8, 13, 21\}$$

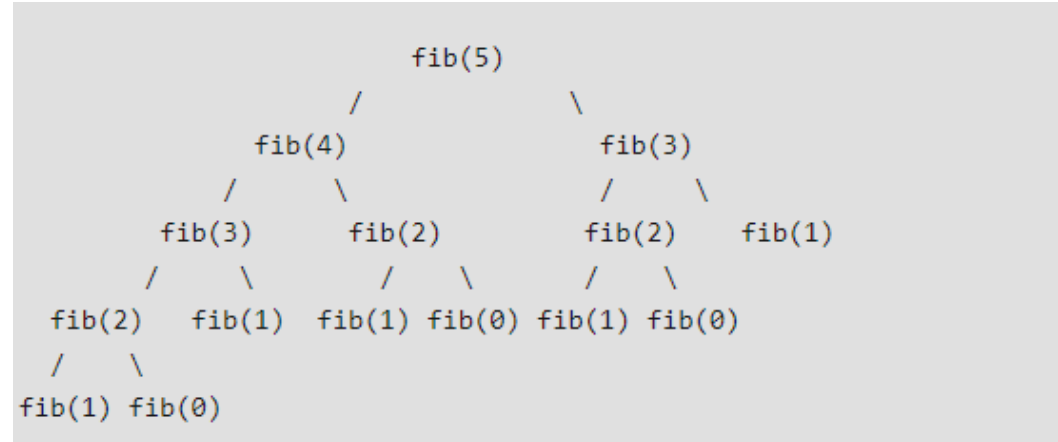
Optimal Substructure

```
/* simple recursive program for Fibonacci numbers */  
int fib(int n)  
{  
    if ( n <= 1 )  
        return n;  
    return fib(n-1) + fib(n-2);  
}
```

Overlapping Subproblems



Time Complexity



- **Recurrent relation**

$$T(n) = T(n - 1) + T(n - 2) + O(1)$$

$$\text{where, } T(n \leq 1) = O(1)$$

$$\text{Intuitively, } T(n) = O(2^n)$$

Solution: Use Memory

“Remember the results.”

Do not solve the same problem again, just recall it from memory.

Solution: Use Memory

“Remember the results.”

Do not solve the same problem again, just recall it from memory.

Two methods of storing the results in memory

- 1. Memoization (Top-Down)**
- 2. Tabulation (Bottom-up)**

What is Dynamic Programming?

- **Dynamic Programming**
 - **Solves a given complex problem by breaking it into subproblems**
 - **Stores the results of subproblems to avoid computing the same results again.**

Dynamic Programming vs. Divide and Conquer

- **Similarities:**

Both paradigms work by combining solutions to sub-problems

- **Differences:**

Dynamic Programming is mainly used when Overlapping Subproblems property is satisfied.

- **Examples:**

Binary Search

Fibonacci Series

Where to use Dynamic Programming?

Following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic Programming:

- 1. Overlapping Subproblems**
- 2. Optimal Substructure**

Performance

Time taken for calculating the 40th Fibonacci number (102334155):

- 1. Recursive: 14 seconds**
- 2. Memoization: 0.17 seconds**
- 3. Tabulation: 0.30 seconds**

Memoization

The Algorithm

- **Initialize a lookup array/table with all its elements as NIL**
 - **NIL is simply a constant value, e.g. -1, that signifies absence of a solution**
- **Call the recursive function $f(n)$ to solve for 'n' using memoization**

The Algorithm

- **At every step i , $f(i)$ performs the following steps:**
 - 1. Checks whether $\text{table}[i]$ is NIL or not.**
 - 2. If it's not NIL, $f(i)$ returns the value ' $\text{table}[i]$ '.**

The Algorithm

- **At every step i , $f(i)$ performs the following steps:**
 - 1. Checks whether $\text{table}[i]$ is NIL or not.**
 - 2. If it's not NIL, $f(i)$ returns the value ' $\text{table}[i]$ '.**
 - 3. If it's NIL and ' i ' satisfies the base condition, we update the lookup table with the base value and return the same.**

The Algorithm

- **At every step i , $f(i)$ performs the following steps:**
 1. **Checks whether $\text{table}[i]$ is NIL or not.**
 2. **If it's not NIL, $f(i)$ returns the value ' $\text{table}[i]$ '.**
 3. **If it's NIL and ' i ' satisfies the base condition, we update the lookup table with the base value and return the same.**
 4. **If it's NIL and ' i ' does not satisfy the base condition, then $f(i)$ splits the problem ' i ' into subproblems and recursively calls itself to solve them.**
 5. **After the recursive calls return, $f(i)$ combines the solutions to subproblems, updates the lookup table and returns the solution for problem ' i '.**

Dry Run for Fibonacci Number

```

              fib(5)
             /    \
          fib(4)    fib(3)
         /  \    /  \
      fib(3) fib(2) fib(2) fib(1)
     /  \  /  \  /  \
  fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)
 /  \
fib(1) fib(0)
```

table[5]	
table[4]	
table[3]	
table[2]	
table[1]	
table[0]	

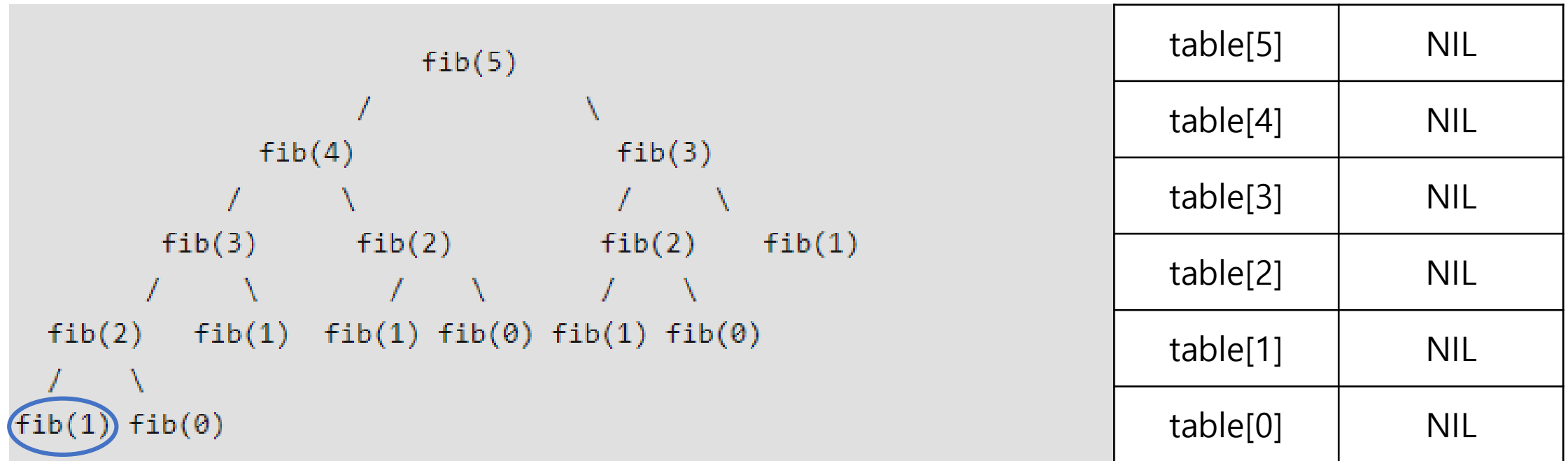
Dry Run for Fibonacci Number

```

              fib(5)
             /    \
          fib(4)    fib(3)
         /  \    /  \
      fib(3) fib(2) fib(2) fib(1)
     /  \  /  \  /  \
  fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)
 /  \
fib(1) fib(0)
```

table[5]	NIL
table[4]	NIL
table[3]	NIL
table[2]	NIL
table[1]	NIL
table[0]	NIL

Dry Run for Fibonacci Number



Dry Run for Fibonacci Number

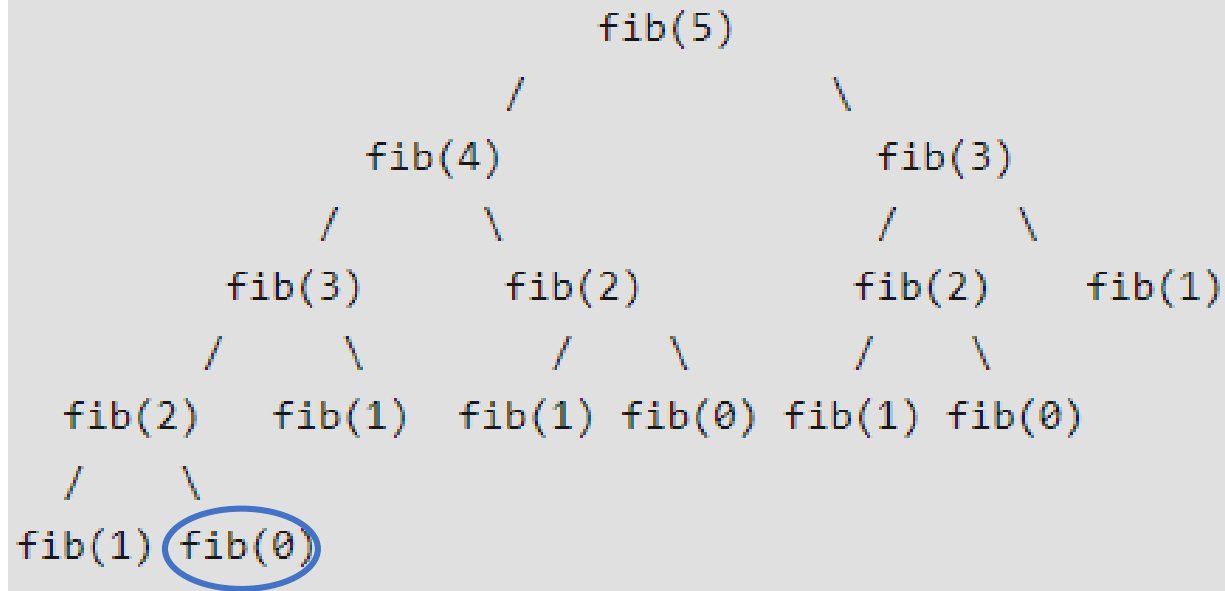
```

              fib(5)
             /   \
          fib(4)   fib(3)
         /  \   /  \
      fib(3) fib(2) fib(2) fib(1)
     /  \   /  \   /  \
  fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)
 /  \
fib(1) fib(0)

```

table[5]	NIL
table[4]	NIL
table[3]	NIL
table[2]	NIL
table[1]	1
table[0]	NIL

Dry Run for Fibonacci Number



table[5]	NIL
table[4]	NIL
table[3]	NIL
table[2]	NIL
table[1]	1
table[0]	NIL

Dry Run for Fibonacci Number

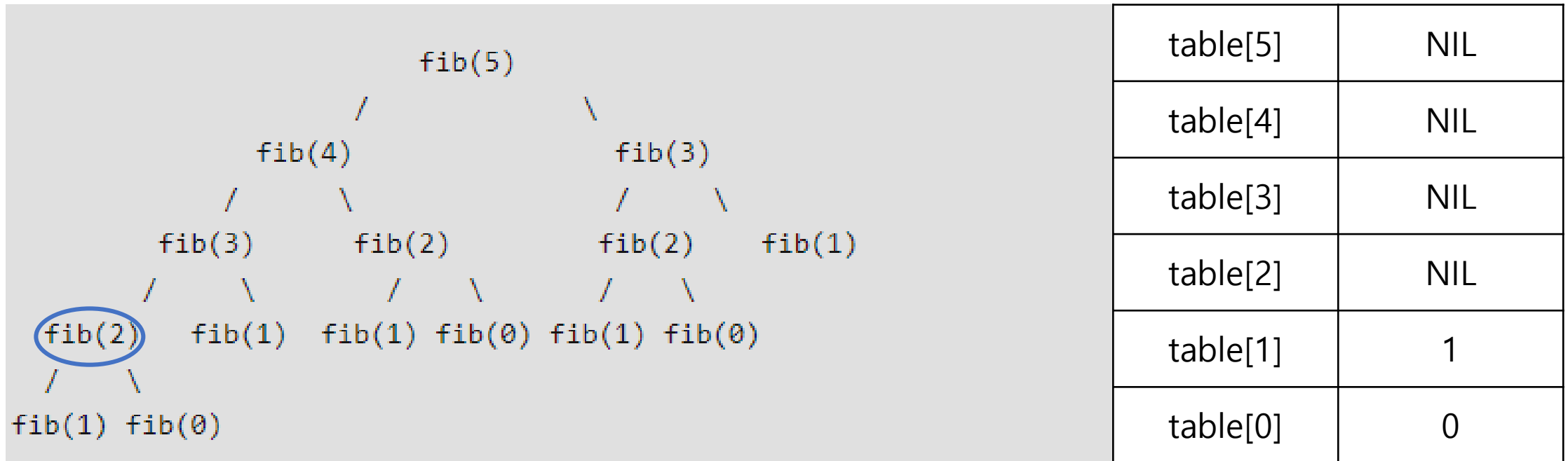
```

                fib(5)
              /      \
            fib(4)    fib(3)
          /   \    /   \
        fib(3) fib(2) fib(2) fib(1)
       /  \   /  \   /  \
    fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)
   /  \
 fib(1) fib(0)

```

table[5]	NIL
table[4]	NIL
table[3]	NIL
table[2]	NIL
table[1]	1
table[0]	0

Dry Run for Fibonacci Number



Dry Run for Fibonacci Number

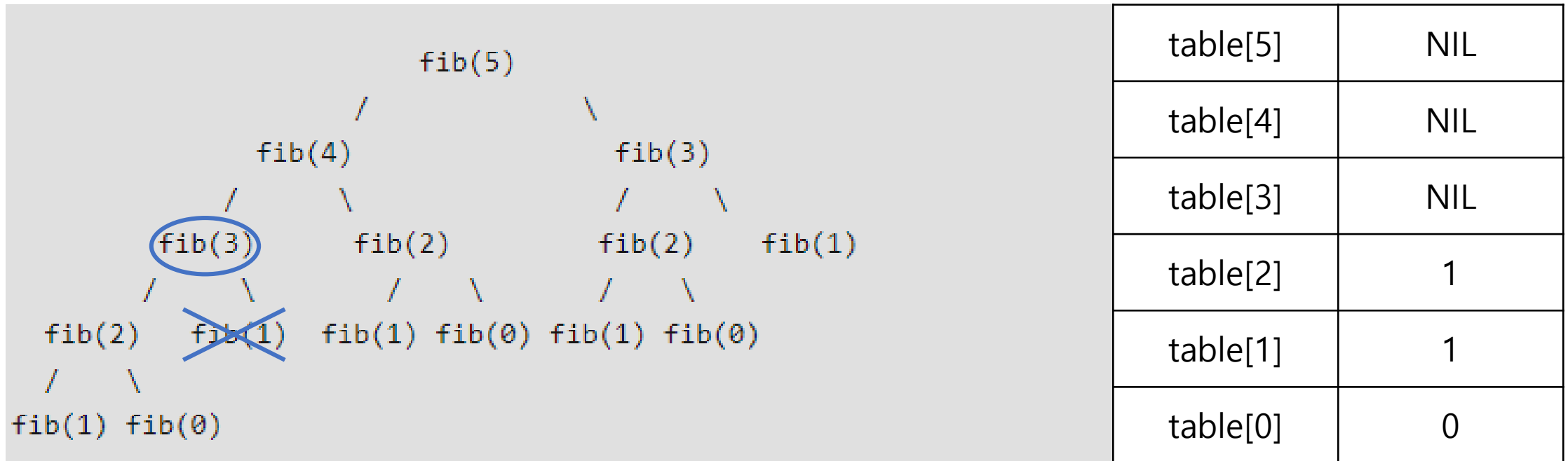
```

      fib(5)
      /   \
    fib(4) fib(3)
    /  \   /  \
  fib(3) fib(2) fib(2) fib(1)
  /  \   /  \   /  \
fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)
/  \
fib(1) fib(0)

```

table[5]	NIL
table[4]	NIL
table[3]	NIL
table[2]	1
table[1]	1
table[0]	0

Dry Run for Fibonacci Number



Dry Run for Fibonacci Number

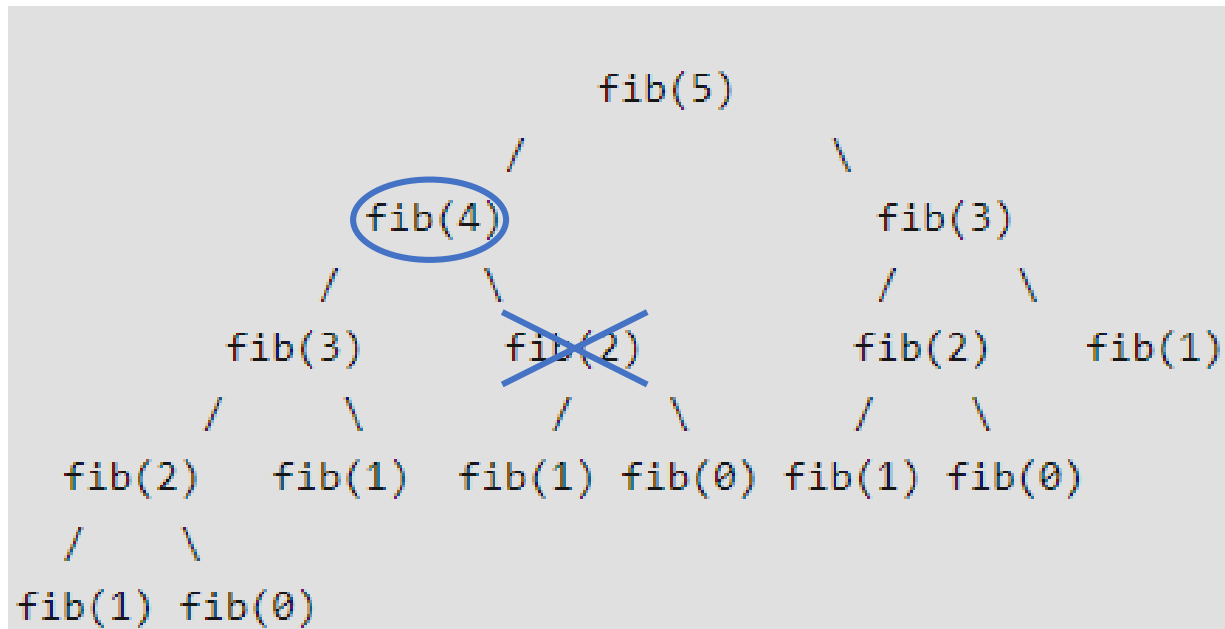
```

                fib(5)
              /      \
            fib(4)    fib(3)
          /   \    /   \
        fib(3) fib(2) fib(2) fib(1)
       /  \   /  \   /  \
    fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)
   /  \
 fib(1) fib(0)

```

table[5]	NIL
table[4]	NIL
table[3]	2
table[2]	1
table[1]	1
table[0]	0

Dry Run for Fibonacci Number



table[5]	NIL
table[4]	NIL
table[3]	2
table[2]	1
table[1]	1
table[0]	0

Dry Run for Fibonacci Number

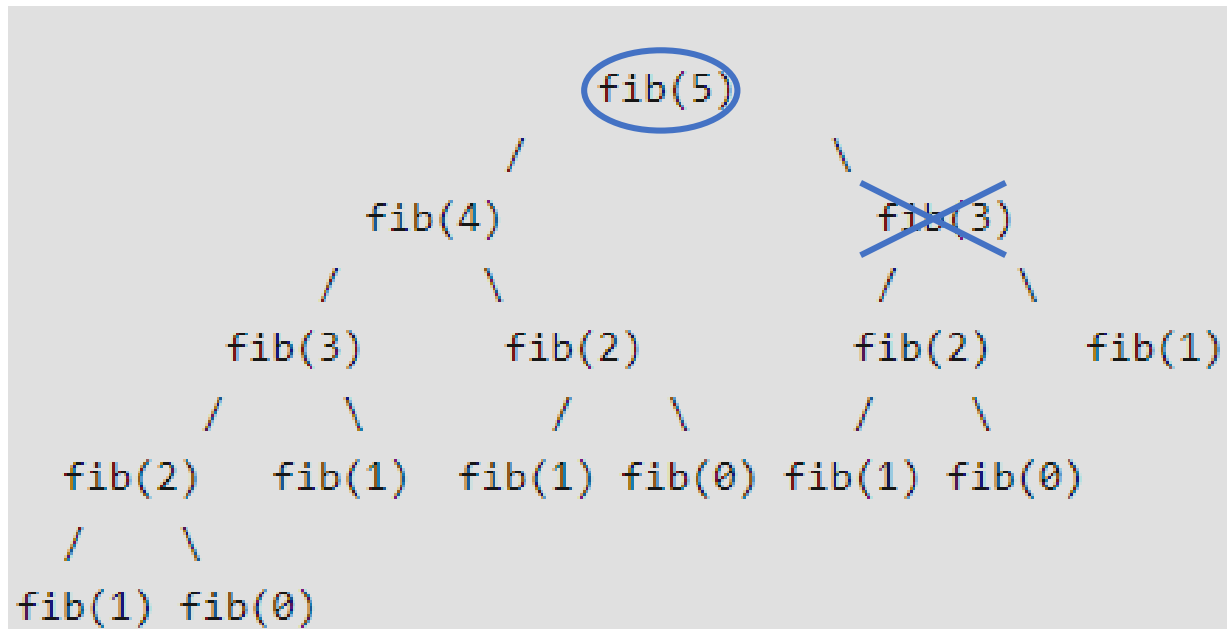
```

              fib(5)
             /    \
          fib(4)    fib(3)
         /  \    /  \
      fib(3) fib(2) fib(2) fib(1)
     /  \  /  \  /  \
  fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)
 /  \
fib(1) fib(0)

```

table[5]	NIL
table[4]	3
table[3]	2
table[2]	1
table[1]	1
table[0]	0

Dry Run for Fibonacci Number



table[5]	NIL
table[4]	3
table[3]	2
table[2]	1
table[1]	1
table[0]	0

Dry Run for Fibonacci Number

```

      fib(5)
      /    \
    fib(4)  fib(3)
    /  \    /  \
  fib(3) fib(2) fib(2) fib(1)
  /  \    /  \    /  \
fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)
/  \
fib(1) fib(0)

```

table[5]	5
table[4]	3
table[3]	2
table[2]	1
table[1]	1
table[0]	0

Dry Run for Fibonacci Number

```
          fib(5)
         /    \
      fib(4)    fib(3)
     /  \    /  \
  fib(3) fib(2) fib(2) fib(1)
 /  \  /  \  /  \
fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)
/  \
fib(1) fib(0)
```

table[5]	5
table[4]	3
table[3]	2
table[2]	1
table[1]	1
table[0]	0

→ Time Complexity: $O(n)$

Code for Fibonacci Number

```
#include <bits/stdc++.h>
using namespace std;
#define NIL -1
#define MAX 100

int lookup[MAX];

/* Function to initialize NIL
values in lookup table */
void _initialize()
{
    int i;
    for (i = 0; i < MAX; i++)
        lookup[i] = NIL;
}

/* function for nth Fibonacci number */
int fib(int n)
{
    if (lookup[n] == NIL)
    {
        if (n <= 1)
            lookup[n] = n;
        else
            lookup[n] = fib(n - 1) + fib(n - 2);
    }
}
```

Tabulation

The Algorithm

- **Build the lookup table in bottom up fashion**
- **After the table is built, simply return table[n]**

The Algorithm

- **Steps:**

1. We begin with initializing the base values of 'i'.
2. Next, we run a loop that iterates over the remaining values of 'i'.
3. At every iteration i , $f(n)$ updates the i th entry in the lookup table by combining the solutions to the previously solved subproblems.
4. Finally, $f(n)$ returns $table[n]$.

Dry Run for Fibonacci Number

```

              fib(5)
             /    \
          fib(4)    fib(3)
         /  \    /  \
      fib(3) fib(2) fib(2) fib(1)
     /  \  /  \  /  \
  fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)
 /  \
fib(1) fib(0)
```

table[5]	
table[4]	
table[3]	
table[2]	
table[1]	
table[0]	

Dry Run for Fibonacci Number

```
          fib(5)
         /    \
      fib(4)    fib(3)
     /  \    /  \
  fib(3) fib(2) fib(2) fib(1)
 /  \  /  \  /  \
fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)
/  \
fib(1) fib(0)
```

table[5]	
table[4]	
table[3]	
table[2]	
table[1]	1
table[0]	0

Dry Run for Fibonacci Number

```

                fib(5)
              /      \
            fib(4)    fib(3)
          /   \    /   \
        fib(3) fib(2) fib(2) fib(1)
       /  \   /  \   /  \
    fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)
   /  \
 fib(1) fib(0)

```

table[5]	
table[4]	
table[3]	
table[2]	1
table[1]	1
table[0]	0

Dry Run for Fibonacci Number

```

                fib(5)
              /      \
            fib(4)    fib(3)
          /   \    /   \
        fib(3) fib(2) fib(2) fib(1)
       /  \   /  \   /  \
    fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)
   /  \
 fib(1) fib(0)

```

table[5]	
table[4]	
table[3]	2
table[2]	1
table[1]	1
table[0]	0

Dry Run for Fibonacci Number

```

              fib(5)
             /    \
          fib(4)    fib(3)
         /  \    /  \
       fib(3) fib(2) fib(2) fib(1)
      /  \  /  \  /  \
    fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)
   /  \
 fib(1) fib(0)

```

table[5]	
table[4]	3
table[3]	2
table[2]	1
table[1]	1
table[0]	0

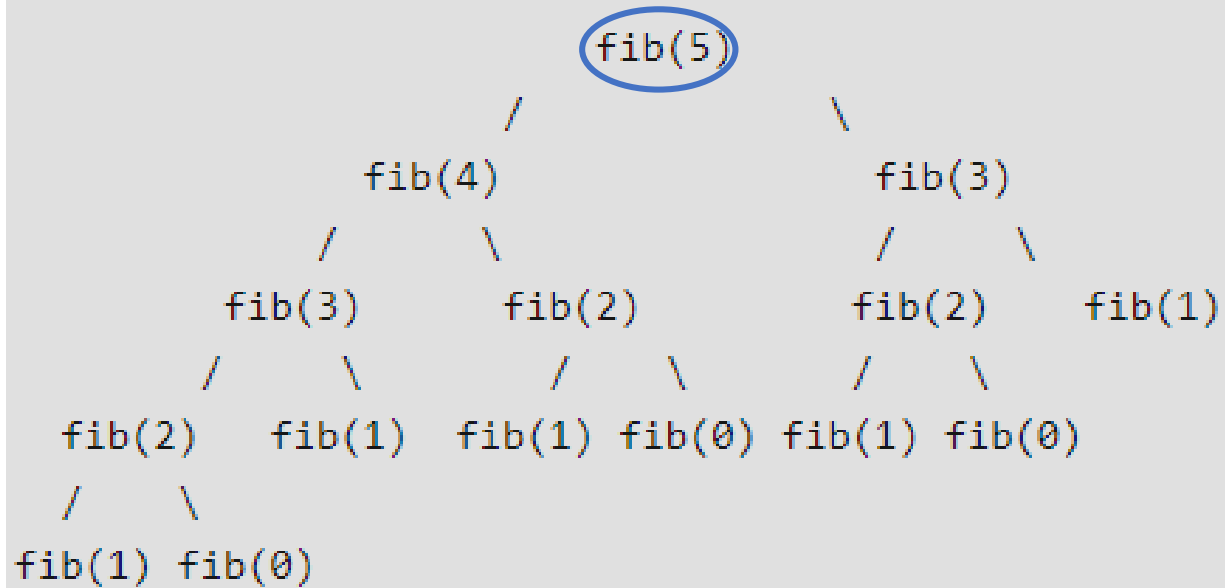
Dry Run for Fibonacci Number

```

          fib(5)
         /    \
      fib(4)    fib(3)
     /  \    /  \
  fib(3) fib(2) fib(2) fib(1)
 /  \  /  \  /  \
fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)
/  \
fib(1) fib(0)
```

table[5]	5
table[4]	3
table[3]	2
table[2]	1
table[1]	1
table[0]	0

Dry Run for Fibonacci Number



table[5]	5
table[4]	3
table[3]	2
table[2]	1
table[1]	1
table[0]	0

Tabulation or Memoization

- **Tabulation**

- Works in bottom up fashion
- Avoids multiple lookups, thus, saves function call overhead time

- **Memoization**

- Works in top down fashion
- Sometimes, avoids computing solutions to subproblems that are not needed, e.g., Longest Common Subsequence
- Sometimes, more intuitive to write, e.g., Matrix Chain Multiplication

Tabulation or Memoization

	Tabulation	Memoization
State	State Transition relation is difficult to think	State transition relation is easy to think
Code	Code gets complicated when lot of conditions are required	Code is easy and less complicated
Speed	Fast, as we directly access previous states from the table	Slow due to lot of recursive calls and return statements
Subproblem solving	If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor	If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required
Table Entries	In Tabulated version, starting from the first entry, all entries are filled one by one	Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version. The table is filled on demand.

Code for Fibonacci Number

```
#include<stdio.h>
int fib(int n)
{
    int f[n+1];
    int i;
    f[0] = 0;    f[1] = 1;
    for (i = 2; i <= n; i++)
        f[i] = f[i-1] + f[i-2];

    return f[n];
}
```

Optimal Substructure

Optimal Substructure

A given problem is said to have the Optimal Substructure property if an optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

Optimal Substructure

For Example,

- **The Shortest Path problem has following Optimal Substructure property:**
 - If a node x lies in the shortest path from source node u to destination node v then, the shortest path from u to v is the combination of shortest path from u to x and shortest path from x to v .
- **All Pair Shortest Path**
 - Floyd-Warshall
 - Bellman-Ford

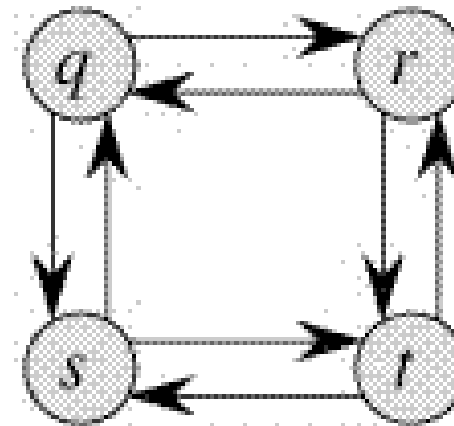
Optimal Substructure

On the other hand,

- **The Longest Path problem doesn't have the Optimal Substructure property.**
- **Here, by Longest Path we mean longest simple path (path without cycle) between two nodes.**

Optimal Substructure

- Longest paths from q to t:
 - $q \rightarrow r \rightarrow t$
 - $q \rightarrow s \rightarrow t$
- Longest paths from q to r:
 - $q \rightarrow s \rightarrow t \rightarrow r$
- Longest paths from q to r:
 - $r \rightarrow q \rightarrow s \rightarrow t$



- The optimal solution to the main problem can not be obtained using optimal solutions to subproblems.

How to solve DP?

How to solve a Dynamic Programming Problem?

Steps to solve a DP

1. Identify if it is a DP problem
2. Decide a state expression with least parameters
3. Formulate state relationship
4. Do tabulation (or add memoization)

How to solve a Dynamic Programming Problem?

Step 1: How to classify a problem as a DP Problem?

- Typically, 1) all the problems that require to maximize or minimize certain quantity or 2) counting problems that say to count the arrangements under certain condition or certain probability problems can be solved by using DP.
- All DP problems satisfy the overlapping subproblems property and most of the classic dynamic problems also satisfy the optimal substructure property.

How to solve a Dynamic Programming Problem?


Step 2: Deciding the state

- DP problems are all about state and their transition. This is the most basic step which must be done very carefully because the state transition depends on the choice of state definition you make.
- State: A state can be defined as the set of parameters that can uniquely identify a certain position or standing in the given problem. This set of parameters should be as small as possible to reduce state space.

How to solve a Dynamic Programming Problem?

Step 2: Deciding the state

```
          fib(5)
        /      \
      fib(4)    fib(3)
     /  \    /  \
  fib(3) fib(2) fib(2) fib(1)
 /  \  /  \  /  \
fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)
/  \
fib(1) fib(0)
```



table[5]	5
table[4]	3
table[3]	2
table[2]	1
table[1]	1
table[0]	0

How to solve a Dynamic Programming Problem?

Step 3: Formulating a relation among the states

- **This part is the hardest part of for solving a DP problem and requires a lot of intuition, observation and practice.**

How to solve a Dynamic Programming Problem?

Step 3: Formulating a relation among the states

Given 3 numbers {1, 3, 5}, we need to tell the total number of ways we can form a number 'N' using the sum of the given three numbers. (allowing repetitions and different arrangements).

Total number of ways to form 6 is: 8

1+1+1+1+1+1

1+1+1+3

1+1+3+1

1+3+1+1

3+1+1+1

3+3

1+5

5+1

How to solve a Dynamic Programming Problem?

Step 3: Formulating a relation among the states

Given 3 numbers {1, 3, 5}, we need to tell the total number of ways we can form a number 'N' using the sum of the given three numbers. (allowing repetitions and different arrangements).

Total number of ways to form 6 is: 8

1+1+1+1+1+1

1+1+1+3

1+1+3+1

1+3+1+1

3+1+1+1

3+3

1+5

5+1

Let us assume that we know the result for $n = 1, 2, 3, 4, 5, 6$

→ Let us say we know the result for state (n=1), state (n=2), state (n=3), state (n = 6)

How to solve a Dynamic Programming Problem?

Step 3: Formulating a relation among the states

Given 3 numbers {1, 3, 5}, we need to tell the total number of ways we can form a number 'N' using the sum of the given three numbers. (allowing repetitions and different arrangements).

Total number of ways to form 6 is: 8

1+1+1+1+1+1

1+1+1+3

1+1+3+1

1+3+1+1

3+1+1+1

3+3

1+5

5+1

Now, we wish to know the result of the state ($n = 7$). See, we can only add 1, 3 and 5.

How to solve a Dynamic Programming Problem?

Step 3: Formulating a relation among the states

Given 3 numbers {1, 3, 5}, we need to tell the total number of ways we can form a number 'N' using the sum of the given three numbers. (allowing repetitions and different arrangements).

Total number of ways to form 6 is: 8

1+1+1+1+1+1

1+1+1+3

1+1+3+1

1+3+1+1

3+1+1+1

3+3

1+5

5+1

Now, we wish to know the result of the state ($n = 7$). See, we can only add 1, 3 and 5.

Now we can get a sum total of 7 by the following 3 ways:

- 1) Adding 1 to all possible combinations of state ($n = 6$)
- 2) Adding 3 to all possible combinations of state ($n = 4$)
- 3) Adding 5 to all possible combinations of state ($n = 2$)

How to solve a Dynamic Programming Problem?

Step 3: Formulating a relation among the states

Given 3 numbers {1, 3, 5}, we need to tell the total number of ways we can form a number 'N' using the sum of the given three numbers. (allowing repetitions and different arrangements).

Total number of ways to form 6 is: 8

1+1+1+1+1+1

1+1+1+3

1+1+3+1

1+3+1+1

3+1+1+1

3+3

1+5

5+1

1) Adding 1 to all possible combinations of state (n = 6):

$[(1+1+1+1+1+1) + 1]$

$[(1+1+1+3) + 1]$

$[(1+1+3+1) + 1]$

$[(1+3+1+1) + 1]$

$[(3+1+1+1) + 1]$

$[(3+3) + 1]$

$[(1+5) + 1]$

$[(5+1) + 1]$

How to solve a Dynamic Programming Problem?

Step 3: Formulating a relation among the states

Given 3 numbers {1, 3, 5}, we need to tell the total number of ways we can form a number 'N' using the sum of the given three numbers. (allowing repetitions and different arrangements).

Total number of ways to form 6 is: 8

1+1+1+1+1+1

1+1+1+3

1+1+3+1

1+3+1+1

3+1+1+1

3+3

1+5

5+1

2) Adding 3 to all possible combinations of state (n = 4):

$[(1+1+1+1) + 3]$

$[(1+3) + 3]$

$[(3+1) + 3]$

How to solve a Dynamic Programming Problem?

Step 3: Formulating a relation among the states

Given 3 numbers {1, 3, 5}, we need to tell the total number of ways we can form a number 'N' using the sum of the given three numbers. (allowing repetitions and different arrangements).

Total number of ways to form 6 is: 8

1+1+1+1+1+1

1+1+1+3

1+1+3+1

1+3+1+1

3+1+1+1

3+3

1+5

5+1

Therefore, we can say that result for $\text{state}(7) = \text{state}(6) + \text{state}(4) + \text{state}(2)$

In general,
 $\text{state}(n) = \text{state}(n-1) + \text{state}(n-3) + \text{state}(n-5)$

How to solve a Dynamic Programming Problem?

Step 3: Formulating a relation among the states

Given 3 numbers {1, 3, 5}, we need to tell the total number of ways we can form a number 'N' using the sum of the given three numbers. (allowing repetitions and different arrangements).

Total number of ways to form 6 is: 8

1+1+1+1+1+1

1+1+1+3

1+1+3+1

1+3+1+1

3+1+1+1

3+3

1+5

5+1

```
int solve(int n)
{
    // base case
    if (n < 0)
        return 0;
    if (n == 0)
        return 1;

    return solve(n-1) + solve(n-3) + solve(n-5);
}
```

How to solve a Dynamic Programming Problem?

Step 4: Adding memoization or tabulation for the state

Given 3 numbers {1, 3, 5}, we need to tell the total number of ways we can form a number 'N' using the sum of the given three numbers. (allowing repetitions and different arrangements).

Total number of ways to form 6 is: 8

1+1+1+1+1+1

1+1+1+3

1+1+3+1

1+3+1+1

3+1+1+1

3+3

1+5

5+1

```
// initialize to -1
int dp[MAXN];

// this function returns the number of
// arrangements to form 'n'
int solve(int n)
{
    // base case
    if (n < 0)
        return 0;
    if (n == 0)
        return 1;

    // checking if already calculated
    if (dp[n] != -1)
        return dp[n];

    // storing the result and returning
    return dp[n] = solve(n-1) + solve(n-3) + solve(n-5);
}
```

Adding memoization

Reference

- Charles Leiserson and Piotr Indyk, “*Introduction to Algorithms*”, September 29, 2004
- <https://www.geeksforgeeks.org>