Problem Solving

Codility and Leetcode Practitioner

“Brute Force” to “Dynamic Programming”

Problems

Description:

Write a function `fib(n)` that takes in a number as an argument and return the n-th number of the Fibonacci sequence.

Example:

- fib(1) is 1
- fib(2) is 1
- fib(3) is 2
- fib(4) is 3
- fib(n) is fib(n-1)+fib(n-2)

Brute-force

Costly but work

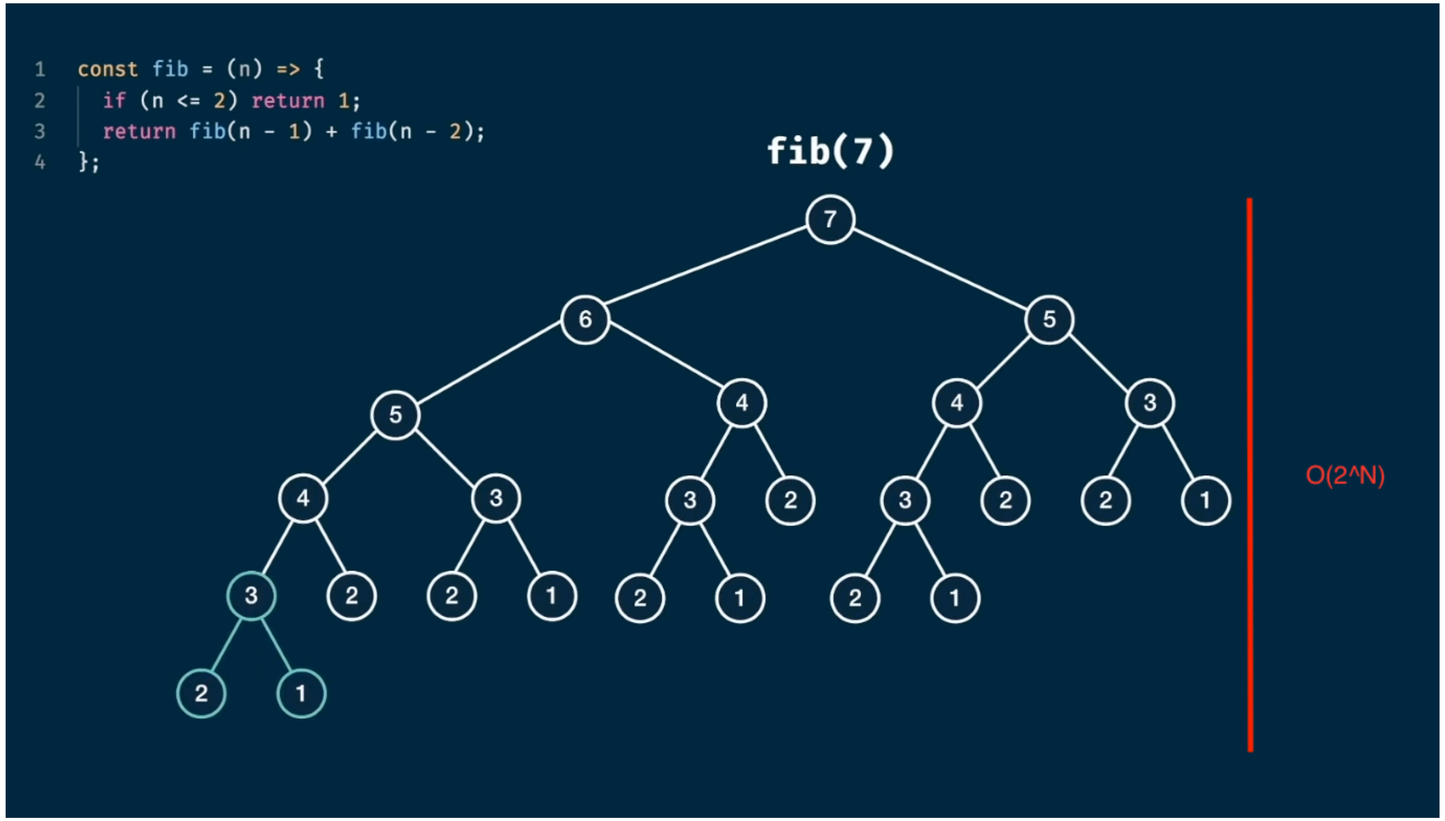
```java
public long fib(int n) {
    if (n <= 2) return 1;

    return fib(n-1) + fib(n-2);
}

public long fibWithoutRecursion(int n) {
    if (n <= 2) return 1;

    Stack<Integer> stack = new Stack<>();
    long result = 0;

    stack.push(n);

    while (!stack.isEmpty()) {
        int current = stack.pop();

        if (current <= 2) {
            result += 1;
        } else {
            // Push both subproblems to stack
            stack.push(current - 1);
            stack.push(current - 2);
        }
    }

    return result;
}
```

Use stack memory
Limited size (1-8Mb)

Use Heap memory
Much larger than stack

Brute-force

Costly but work



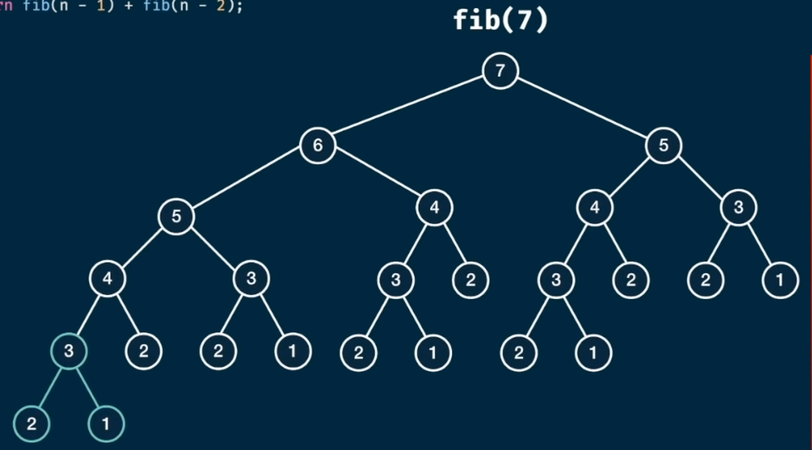
Memoization

Remove duplicate by Caching

```java
public long fib(int n) {  2 usages  new *
    if (n <= 2) return 1L;

    return fib( n: n - 1) + fib( n: n - 2);
}

public long fibWithMem(int n, Map<Integer, Long> mem) {  2 usages  new *
    if (mem.containsKey(n)) {
        return mem.get(n);
    }

    if (n <= 2) {
        mem.put(n, 1L);
        return 1L;
    }

    long result = fibWithMem( n: n - 1, mem) + fibWithMem( n: n - 2, mem);
    mem.put(n, result);

    return result;
}
```

Brute-force (2^N)

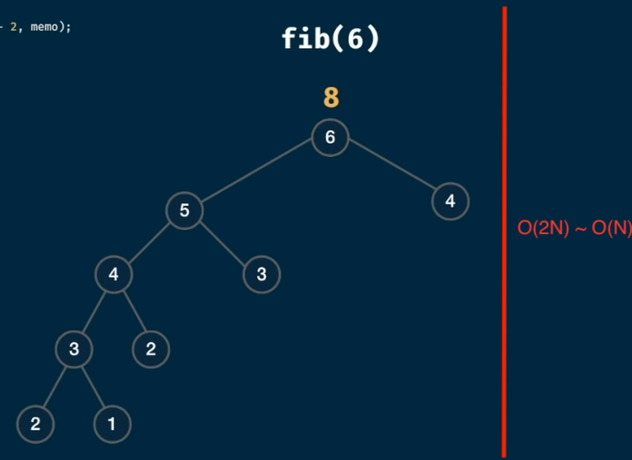
```
1  const fib = (n) => {
2    if (n <= 2) return 1;
3    return fib(n - 1) + fib(n - 2);
4  };
```



Memoization (2N)

```
1  const fib = (n, memo = {}) => {
2    if (n in memo) return memo[n];
3    if (n <= 2) return 1;
4
5    memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
6    return memo[n];
7  };
```

```
memo
{
 3: 2,
 4: 3,
 5: 5,
 6: 8
}
```



Dynamic Programming

Solving Big Problems by Reusing Small Ones

```java
public long dynamicProgramming(int n) {  7 usages  new *
    if (n <= 2) return 1;

    long[] dp = new long[n + 1];
    dp[1]=1;
    dp[2]=1;

    for (int i = 3; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }

    return dp[n];
}

public long dynamicProgrammingWithSlidingWindow(int n) {  7 usages  new *
    if (n <= 2) return 1;
    long n1 = 1L, n2 = 1L;

    for (int i = 3; i <= n; i++) {
        long temp = n1;
        n1 += n2;
        n2 = temp;
    }

    return n1;
}
```

Problems

Description:

Given a natural number n <= 100. How many ways are there to partition n into a sum of positive integers? Permutations of each partition are considered the same way.

Example:

If n = 4, there are 5 ways. Notice that 3 + 1 is the same as 1 + 3

- 4
- 3+1
- 2+2
- 2+1+1
- 1+1+1+1

Brute-force

```java
public long bruteForce(int n) {  4 usages  new *
    if (n == 0) {
        return 1;
    }

    return bruteForce(n, n);
}

public long bruteForce(int n, int maxVal) {  3 usages  new *
    if (n == 0) return 1;

    if (n < 0) return 0;

    if (maxVal == 0) {
        return 0;
    }

    long waysIncludingMaxVal = bruteForce( n: n - maxVal, maxVal);
    long waysExcludeMaxVal = bruteForce(n,  maxVal: maxVal - 1);

    return waysIncludingMaxVal + waysExcludeMaxVal;
}
```

Brute-force

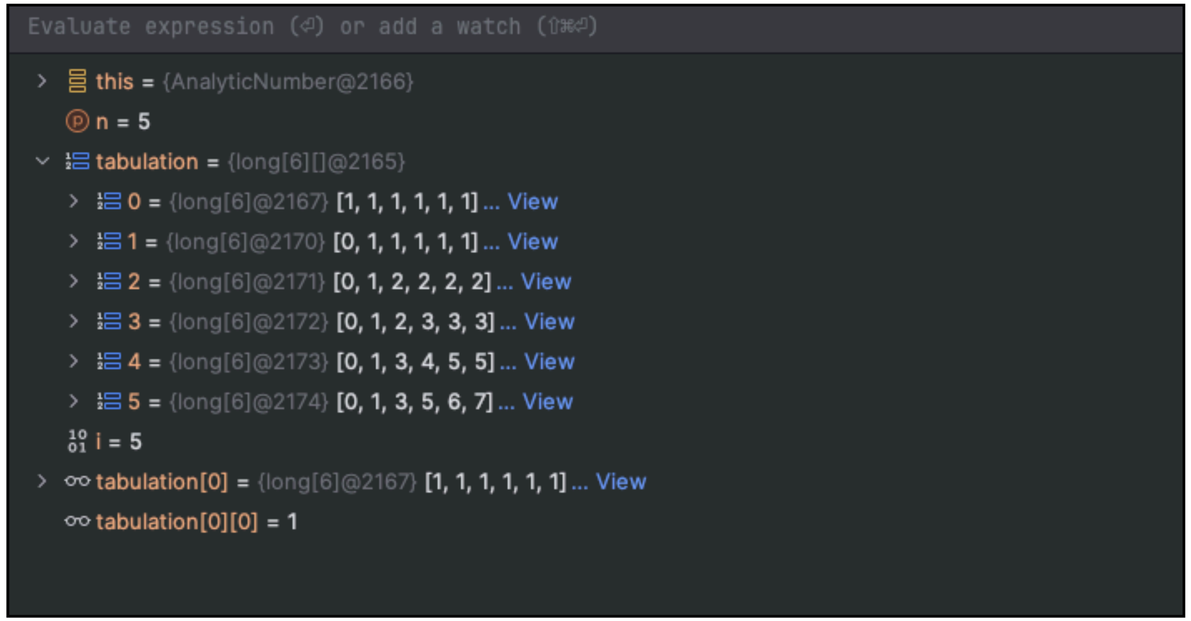
With Memo

```java
public long bruteForceWithMemo(int n, int maxVal, Map<String, Long> memo) {  3 usages  new *
    recursiveCount++;
    if (n == 0) return 1;

    if (n < 0) return 0;

    if (maxVal == 0) {
        return 0;
    }

    String memoKey = String.format("%d-%d", n, maxVal);
    if (memo.containsKey(memoKey)) {
        duplicateBranchCutted++;
        return memo.get(memoKey);
    }

    long waysIncludingMaxVal = bruteForceWithMemo( n: n - maxVal, maxVal, memo);
    long waysExcludeMaxVal = bruteForceWithMemo(n,  maxVal: maxVal - 1, memo);

    memo.put(memoKey, waysIncludingMaxVal + waysExcludeMaxVal);

    return waysIncludingMaxVal + waysExcludeMaxVal;
}
```

Dynamic Programming

With Tabulation

```java
public long dynamicProgrammingWithTabulation(int n) {  4 usages  new *
    long[][] tabulation = new long[n + 1][n + 1];
    tabulation[0][0] = 1;

    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= n; j++) {
            long waysIncludingMaxVal = (i - j < 0) ? 0 : tabulation[i - j][j];
            long waysExcludeMaxVal = (j - 1 < 0) ? 0 : tabulation[i][j - 1];
            tabulation[i][j] = waysIncludingMaxVal + waysExcludeMaxVal;
        }
    }

    return tabulation[n][n];
}
```



Dynamic Programming

From small to big

```java
public long dynamicProgrammingWithTabulation(int n) {  4 usages  new *
    long[][] tabulation = new long[n + 1][n + 1];
    tabulation[0][0] = 1;

    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= n; j++) {
            long waysIncludingMaxVal = (i - j < 0) ? 0 : tabulation[i - j][j];
            long waysExcludeMaxVal = (j - 1 < 0) ? 0 : tabulation[i][j - 1];
            tabulation[i][j] = waysIncludingMaxVal + waysExcludeMaxVal;
        }
    }

    return tabulation[n][n];
}
```

Start from smallest
Scale to destination,
Caching the result to
tabulation

Brute force

From big to small

```java
public long bruteForce(int n) {  4 usages  new *
    if (n == 0) {
        return 1;
    }

    return bruteForce(n, n);
}

public long bruteForce(int n, int maxVal) {  3 usages  new *
    if (n == 0) return 1;

    if (n < 0) return 0;

    if (maxVal == 0) {
        return 0;
    }

    long waysIncludingMaxVal = bruteForce( n: n - maxVal, maxVal);
    long waysExcludeMaxVal = bruteForce(n,  maxVal: maxVal - 1);

    return waysIncludingMaxVal + waysExcludeMaxVal;
}
```

Start from destination
Breakdown to smallest
and return.
Wasted time on rework

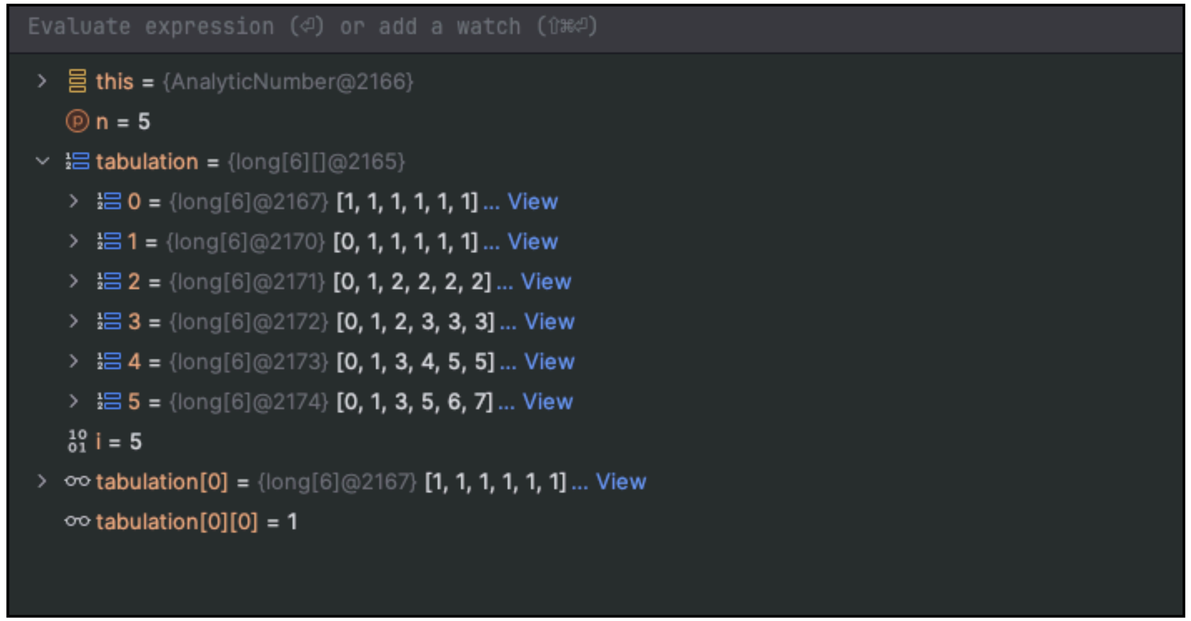
Run Code

Run test and compare the run times

Dynamic Programming

With Tabulation (Space complexity is O(n^2) - 2D)

```java
public long dynamicProgrammingWithTabulation(int n) {  4 usages  new *
    long[][] tabulation = new long[n + 1][n + 1];
    tabulation[0][0] = 1;

    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= n; j++) {
            long waysIncludingMaxVal = (i - j < 0) ? 0 : tabulation[i - j][j];
            long waysExcludeMaxVal = (j - 1 < 0) ? 0 : tabulation[i][j - 1];
            tabulation[i][j] = waysIncludingMaxVal + waysExcludeMaxVal;
        }
    }

    return tabulation[n][n];
}
```



Can we somehow convert 2D to 1D - O(n) for space complexity?

Dynamic Programming

Summary

1. How to define state: What parameters define a subproblem?
2. How to handle base case (smallest case)
3. How to write tradition function
4. How to reduce space complexity (Use 1D instead of 2D)

Learn more about DP

https://youtu.be/oBt53YbR9Kk?si=07UnEOktFlSjLKVV

Thank you

It is not the end, It’s just the beginning