**[509 Leetcode Fibonacci](https://leetcode.com/problems/fibonacci-number/description/)**

**🚦 How to Identify a DP Problem (Beginner-Friendly Clues)**

| **Signal from the Problem** | **What it Means** |
| --- | --- |
| Repeated calculations | You are doing the same work again. |
| Recursive formula | E.g., F(n) = F(n-1) + F(n-2) |
| Optimal solution built from smaller subproblems | Classic DP style |
| You’re solving for a number, path, or combination based on previous values | Often DP |

For Fibonacci:

* **It asks for the nth value based on previous values.**
* It has **overlapping subproblems**: to find F(n), you need F(n-1) and F(n-2).
* This is the key characteristic of DP.

**✅ Step 1: Brute Force Recursion**

This approach simply follows the definition:

python

CopyEdit

def fib(n):

if n == 0:

return 0

if n == 1:

return 1

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

**📊 Time Complexity: O(2^n)**

**🧠 Space Complexity: O(n) due to recursive call stack**

**🔍 Dry Run: fib(4)**

fib(4)

= fib(3) + fib(2)

= (fib(2) + fib(1)) + (fib(1) + fib(0))

= ((fib(1) + fib(0)) + 1) + (1 + 0)

= ((1 + 0) + 1) + (1 + 0)

= (1 + 1) + 1 = 2 + 1 = 3

**⚠️ Common Mistakes**

* Infinite recursion (forgetting the base case if n == 0 or n == 1)
* Stack overflow for large n (this is **not efficient** for n > 30)

**❓ Why Does Recursion Cause Stack Overflow for Large n in Fibonacci?**

**1. What is Stack Overflow?**

When you use recursion, **each function call is placed on the call stack**. The stack holds temporary data like:

* Return addresses
* Function parameters
* Local variables

If you make **too many recursive calls**, the stack becomes too full — this is called a **stack overflow**.

**2. How Does This Happen in Fibonacci?**

Let’s look at this code again:

def fib(n):

if n == 0:

return 0

if n == 1:

return 1

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

Each call to fib(n) leads to **two more calls**:

* fib(n - 1)
* fib(n - 2)

So the number of calls **grows exponentially**:

fib(5) = fib(4) + fib(3)

= (fib(3) + fib(2)) + (fib(2) + fib(1))

...

For fib(30), the function is called **over 1 million times**.  
For fib(40), it's called **over 100 million times**!

The call stack can't handle this many nested function calls — it **runs out of memory**, and the program crashes with a **stack overflow error**.

**3. Call Stack Depth**

The **maximum recursion depth in Python** is usually around **1000 by default**. You can check this using:

import sys

print(sys.getrecursionlimit()) # usually prints 1000

So even **before** n gets very large, say n = 1000, the program will crash **not because of time**, but because Python can’t keep track of that many recursive layers on the stack.

**🔥 Key Point**

| **Factor** | **Brute Recursive Fibonacci** |
| --- | --- |
| Number of calls | Exponential (2^n) |
| Stack depth needed | Linear (n) |
| Python max depth | ~1000 |
| Practical max n | ~30–35 before crashing |

**🛡️ How to Avoid Stack Overflow?**

Use **memoization**, **iteration**, or **space-optimized DP** — all avoid deep recursion

**✅ Step 2: Memoization (Top-Down DP)**

We store the values we've already calculated.

def fib(n, memo=None):

if memo is None:

memo = [-1] \* (n + 1)

if n == 0:

return 0

if n == 1:

return 1

if memo[n] != -1:

return memo[n]

memo[n] = fib(n - 1, memo) + fib(n - 2, memo)

return memo[n]

**📊 Time: O(n)**

**🧠 Space: O(n) (due to memo + recursion stack)**

**🔍 Dry Run: fib(4)**

memo = [-1, -1, -1, -1, -1]

fib(4):

fib(3):

fib(2):

fib(1) = 1

fib(0) = 0

memo[2] = 1

fib(1) = 1

memo[3] = 2

fib(2) = memo[2] = 1

memo[4] = 3

Final memo: [-1, -1, 1, 2, 3]

**⚠️ Mistakes to Avoid**

* Forgetting to initialize memo properly
* Recreating memo every call
* Not reusing memoized results

**✅ Step 3: Tabulation (Bottom-Up DP)**

Build the solution iteratively.

def fib(n):

if n == 0:

return 0

dp = [0] \* (n + 1)

dp[0] = 0

dp[1] = 1

for i in range(2, n + 1):

dp[i] = dp[i - 1] + dp[i - 2]

return dp[n]

**📊 Time: O(n)**

**🧠 Space: O(n)**

**🔍 Dry Run: fib(4)**

kotlin

CopyEdit

dp[0] = 0

dp[1] = 1

dp[2] = 1 (dp[1] + dp[0])

dp[3] = 2 (dp[2] + dp[1])

dp[4] = 3 (dp[3] + dp[2])

return dp[4] = 3

**⚠️ Mistakes**

* Index out of bounds: ensure array size is n+1
* Starting loop from wrong index (should be 2)

**🎯 Top-Down vs Bottom-Up DP**

**Dynamic Programming is about breaking a big problem into smaller subproblems, solving them, and storing the results so you don’t repeat work.**

**There are two main ways to do that:**

**✅ 1. Top-Down DP (Recursion + Memoization)**

**💡 How It Works:**

* **Start from the original problem (n) and recursively break it down into smaller subproblems.**
* **Use a memo (cache) to store already computed results.**
* **The recursion stops at base cases, and results "bubble back up."**

**🔧 Example (Fibonacci):**

**python**

**CopyEdit**

**def fib(n, memo=None):**

**if memo is None:**

**memo = [-1] \* (n + 1)**

**if n == 0: return 0**

**if n == 1: return 1**

**if memo[n] != -1:**

**return memo[n]**

**memo[n] = fib(n - 1, memo) + fib(n - 2, memo)**

**return memo[n]**

**📦 Characteristics:**

| **Feature** | **Description** |
| --- | --- |
| **Method** | **Recursive** |
| **Control flow** | **Starts from big problem → small** |
| **Needs memo table?** | **✅ Yes** |
| **Uses stack space?** | **✅ Yes (due to recursion)** |
| **Easy to write?** | **✅ Often more intuitive** |

**✅ 2. Bottom-Up DP (Tabulation)**

**💡 How It Works:**

* **Start from the smallest subproblem, solve each one iteratively, and build up to the answer.**
* **Use an array (dp) to store solutions.**
* **No recursion needed.**

**🔧 Example (Fibonacci):**

**def fib(n):**

**if n == 0: return 0**

**dp = [0] \* (n + 1)**

**dp[0], dp[1] = 0, 1**

**for i in range(2, n + 1):**

**dp[i] = dp[i - 1] + dp[i - 2]**

**return dp[n]**

**📦 Characteristics:**

| **Feature** | **Description** |
| --- | --- |
| **Method** | **Iterative** |
| **Control flow** | **Starts from small → big** |
| **Needs memo table?** | **✅ Yes (usually a dp[] array)** |
| **Uses stack space?** | **❌ No recursion stack used** |
| **Easy to write?** | **❌ Slightly less intuitive** |

**📊 Comparison Table**

| **Feature** | **Top-Down (Memoization)** | **Bottom-Up (Tabulation)** |
| --- | --- | --- |
| **Approach** | **Recursive** | **Iterative** |
| **Stack Space** | **Yes (can cause overflow)** | **No** |
| **Memoization** | **Explicit (dictionary/list)** | **Implicit in loop** |
| **Base case** | **Stops at base** | **Starts from base** |
| **Ease of writing** | **Often easier** | **Slightly more setup** |
| **Performance** | **Similar (O(n) here)** | **Slightly better in practice** |

**🧠 In Summary:**

**| Think of it like this: |**

* **Top-Down: “I want to solve fib(n). Oh wait, I need fib(n-1) and fib(n-2) first.”**
* **Bottom-Up: “Let me solve fib(0), fib(1), then build up until fib(n).”**

**✅ Step 4: Space Optimized DP**

We don’t need the whole array—just the last two values.

def fib(n):

if n == 0:

return 0

prev2 = 0

prev1 = 1

for \_ in range(2, n + 1):

curr = prev1 + prev2

prev2 = prev1

prev1 = curr

return prev1

**📊 Time: O(n)**

**🧠 Space: O(1)**

**🔍 Dry Run: fib(4)**

prev2 = 0

prev1 = 1

i = 2: curr = 1 → prev2 = 1, prev1 = 1

i = 3: curr = 2 → prev2 = 1, prev1 = 2

i = 4: curr = 3 → prev2 = 2, prev1 = 3

return prev1 = 3

**At the end of the loop, prev1 = 5, which is fib(5).**

* **curr is only temporary inside the loop.**
* **After the last loop step (i = n), prev1 holds the result.**

**✅ Conclusion:**

* **prev1 is updated after computing curr, so it ends up holding the latest Fibonacci number.**
* **Returning curr would also work, but only if the loop has run at least once.**
* **Returning prev1 is safer and clearer, especially since it holds the correct value even when n = 1.**

**⚠️ Common Mistake:**

**If someone says "I'll return curr instead" and forgets to handle n = 1 or doesn’t run the loop, the code might crash or return the wrong value.**

**✅ Step 5: Matrix Exponentiation (Advanced)**

Fastest way to compute large Fibonacci numbers (n > 10^5) using matrix power in O(log n).

Skip this for now if you're new to DP — learn it later as you get more comfortable.

**🧠 Summary Table (for Review)**

| **Approach** | **Time** | **Space** | **Use When** |
| --- | --- | --- | --- |
| Recursion | O(2^n) | O(n) | For understanding only |
| Memoization | O(n) | O(n) | When recursion is required |
| Tabulation | O(n) | O(n) | Iterative and readable |
| Space Optimized | O(n) | O(1) | Best practical solution |
| Matrix Power | O(log n) | O(1) | Very large n (advanced topics) |

**✅ Best for Interviews**

def fib(n):

if n == 0:

return 0

prev2, prev1 = 0, 1

for \_ in range(2, n + 1):

curr = prev1 + prev2

prev2, prev1 = prev1, curr

return prev1

[**198 Leetcode House Robber**](https://leetcode.com/problems/house-robber/description/)

**🎯 Problem Recap**

You're given a list of non-negative integers nums, where nums[i] is the amount of money in the i-th house.  
**You cannot rob two adjacent houses.**  
Return the **maximum amount** of money you can rob **without triggering alarms**.

**✅ Approach 1: Brute Force (Recursion)**

**💡 Explanation**

At every index i, we make a choice:

1. **Skip the current house** → rob from i + 1
2. **Rob the current house** → add nums[i] and rob from i + 2

We explore all possibilities recursively.

**🔁 Code (Python)**

python

CopyEdit

def rob(nums):

def robFrom(i):

if i >= len(nums):

return 0

return max(

robFrom(i + 1), # skip current house

nums[i] + robFrom(i + 2) # rob current house

)

return robFrom(0)

**🧪 Dry Run: nums = [2, 7, 9]**

lua

CopyEdit

robFrom(0) = max(robFrom(1), 2 + robFrom(2))

robFrom(1) = max(robFrom(2), 7 + robFrom(3))

robFrom(2) = max(robFrom(3), 9 + robFrom(4))

...

Final result: max(robFrom(1), 2 + robFrom(2)) = max(robFrom(1), robFrom(2) + 2)

**📊 Time & Space Complexity**

| **Metric** | **Value** |
| --- | --- |
| Time | O(2^n) |
| Space | O(n) (stack due to recursion) |

**⚠️ Cons**

* Extremely slow for large n
* Many repeated subproblems
* Not scalable

**✅ Use When:**

* You are learning recursion
* You want to understand the problem structure

**✅ Approach 2: Bottom-Up DP (Tabulation)**

**💡 Explanation**

We use a dp array to store:

* dp[i] = max money that can be robbed starting from house i

We build this from the end towards the front.

**🔁 Code (Python)**

python

CopyEdit

def rob(nums):

n = len(nums)

if n == 0: return 0

if n == 1: return nums[0]

dp = [0] \* (n + 2) # Avoid index error for i+2

for i in range(n - 1, -1, -1):

dp[i] = max(dp[i + 1], nums[i] + dp[i + 2])

return dp[0]

**🧪 Dry Run: nums = [2, 7, 9, 3, 1]**

| **i** | **nums[i]** | **dp[i + 1]** | **dp[i + 2]** | **dp[i]** |
| --- | --- | --- | --- | --- |
| 4 | 1 | 0 | 0 | max(0, 1 + 0) = 1 |
| 3 | 3 | 1 | 0 | max(1, 3 + 0) = 3 |
| 2 | 9 | 3 | 1 | max(3, 9 + 1) = 10 |
| 1 | 7 | 10 | 3 | max(10, 7 + 3) = 10 |
| 0 | 2 | 10 | 10 | max(10, 2 + 10) = **12** |

✅ Final answer: 12

**📊 Time & Space Complexity**

| **Metric** | **Value** |
| --- | --- |
| Time | O(n) |
| Space | O(n) |

**✅ Use When:**

* You want a clear and iterative solution
* Memory usage isn't a problem

**✅ Approach 3: Space-Optimized Bottom-Up DP (Best Practice)**

**💡 Explanation**

Since dp[i] only uses dp[i+1] and dp[i+2], we don’t need the full array — just keep **2 variables**.

**🔁 Code (Python)**

python

CopyEdit

def rob(nums):

prev1 = 0 # dp[i+1]

prev2 = 0 # dp[i+2]

for i in reversed(range(len(nums))):

curr = max(prev1, nums[i] + prev2)

prev2 = prev1

prev1 = curr

return prev1

**🧪 Dry Run: nums = [2, 7, 9, 3, 1]**

| **i** | **nums[i]** | **prev1 (dp[i+1])** | **prev2 (dp[i+2])** | **curr** |
| --- | --- | --- | --- | --- |
| 4 | 1 | 0 | 0 | max(0, 1 + 0) = 1 |
| 3 | 3 | 1 | 0 | max(1, 3 + 0) = 3 |
| 2 | 9 | 3 | 1 | max(3, 9 + 1) = 10 |
| 1 | 7 | 10 | 3 | max(10, 7 + 3) = 10 |
| 0 | 2 | 10 | 10 | max(10, 2 + 10) = **12** |

✅ Final answer: 12

**📊 Time & Space Complexity**

| **Metric** | **Value** |
| --- | --- |
| Time | O(n) |
| Space | O(1) |

**✅ Why This is the Best**

* No extra memory
* Fast
* Simple to implement
* Ideal for large inputs

**🧾 Summary Comparison Table**

| **Approach** | **Time** | **Space** | **Pros** | **When to Use** |
| --- | --- | --- | --- | --- |
| Brute Force | O(2^n) | O(n) | Easy to write, helps understand | Learning recursion |
| Bottom-Up DP (Tabular) | O(n) | O(n) | Clear logic, efficient | General use |
| Space-Optimized Bottom-Up | O(n) | O(1) | ✅ Best practical & interview use | Always (unless tracking path) |