

```
import java.util.Arrays;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionException;
import java.util.concurrent.TimeUnit;
import java.util.Scanner;

public class FibonacciExample {

    public static long fiboAt(int position) {
        /*
         * Approach -> Top-Down (Recursive)
         * Time Complexity -> O(2^N)
         * Space Complexity -> O(N) for the recursion stack.
         * Observation -> Multiple recursive calls with same arguments for some
         * positions.
        */
        validatePosition(position);
        if (position == 0 || position == 1) return position; // Base case.
        return fiboAt(position - 1) + fiboAt(position - 2);
    }

    public static long fiboAt(int position, long[] memory /* Defines the Fibonacci
    number at different positions (/ indices). */) {
        /*
         * Approach -> Memorization, Top-Down (Recursive)
         * Time Complexity -> O(N)
         * Space Complexity -> O(N) - O(N) for recursion stack at worst case + O(N) for
         * memory.
         * Observation -> Improved Time Complexity.
        */
        validatePosition(position);

        if (position == 0 || position == 1) return position; // Base case.

        if (memory == null) {
            memory = new long[position + 1]; // As we are using '0' based indexing, the
            memory must be of size (position + 1).
            Arrays.fill(memory, Long.MIN_VALUE);
        }
    }
}
```

```
        memory[0] = 0; // Not required. Just for theoretical correctness.
        memory[1] = 1; // Not required. just for theoretical correctness.
    }

    if (memory[position] == Long.MIN_VALUE) {
        memory[position] = fiboAt(position - 1, memory) + fiboAt(position - 2, memory);
    }

    return memory[position];
}

public static long fiboDP(int position) {
    /*
     * Approach -> Dynamic Programming (Tabulation), Bottom-Up (Iterative)
     * Time Complexity -> O(N)
     * Space Complexity -> O(N) - O(N) for DP state memory.
     * Observation -> Both Time and Space Complexity is the same as of
     Memorization but,
     the use of recursion stack has been eliminated as stack has very little
     memory
     thus, avoiding the stack overflow.
    */
}

// DP State: dp[i] -> Fibonacci number at 'ith' position.
// As array uses '0' based indexing, fibonacci number for Nth position will be at
Nth index and the size
// of DP state array will be (N + 1).

validatePosition(position);

if (position == 0 || position == 1) return position; // Base case.

long[] dp = new long[position + 1];

dp[0] = 0;
dp[1] = 1;

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

```
        return dp[position];
    }

    public static long fiboDPSO(int position) {
        /*
         * Approach -> Dynamic Programming (Space optimized), Bottom-Up (Iterative)
         * Time Complexity -> O(N)
         * Space Complexity -> O(1)
         * Observation -> Improved Space Complexity as compared to Tabulation.
         */

        // DP State: dp[i] -> Fibonacci number at 'ith' position.
        // As array uses '0' based indexing, fibonacci number for Nth position will be at
        Nth index and the size
        // of DP state array will be (N + 1).

        validatePosition(position);

        if (position == 0 || position == 1) return position; // Base case

        long previousToPrevious = 0;
        long previous = 1;
        long current = 1;

        for (int i = 2; i <= position; i++) {
            current = previous + previousToPrevious;
            previousToPrevious = previous;
            previous = current;
        }

        return current;
    }

    /* Input Validation Function: Throwing error if the position of the Fibonacci number
is negative... */
    public static void validatePosition(int position) {
        if (position < 0) throw new IllegalArgumentException("Position of a number in the
Fibonacci series cannot be negative!");
    }
}
```

```
}

/* Utility Function: Decoding the error message if any error caused during runtime...
*/
public static String decodeException(CompletionException exception) {
    return exception.getMessage();
}

/* Main Function */
public static void main (String[] args) {

    System.out.println("\n--- Finding the 'Nth' Fibonacci Number ---\n");

    System.out.print("Enter the position of the fibonacci number: ");
    java.util.Scanner inputScanner = new java.util.Scanner(System.in); // Opening the
    input...
    int position = inputScanner.nextInt(); // Position of the fibonacci number / Nth
    fibonacci number. (Taking input...)
    inputScanner.close(); // Closing the input...

    // Output for Recursive Algorithm...
    try {
        System.out.print("OUTPUT (Recursive Algorithm) -> ");
        System.out.print(
            CompletableFuture.supplyAsync(() -> fiboAt(position)) // Function Call...
            .orTimeout(10, TimeUnit.SECONDS)
            .join()
            + "\n"
        );
    }
    catch (CompletionException exception) {
        System.out.print(decodeException(exception) + "\n");
    }

    // Output for Memorization Algorithm...
    try {
        System.out.print("OUTPUT (Memorization Algorithm) -> ");
        System.out.print(
            CompletableFuture.supplyAsync(() -> fiboAt(position, null)) // Function Call...
            .orTimeout(10, TimeUnit.SECONDS)
            .join()
        );
    }
}
```

```
        + "\n"
    );
}
catch (CompletionException exception) {
System.out.print(decodeException(exception) + "\n"); }

// Output for Tabulation Algorithm...
try {
System.out.print("OUTPUT (Tabulation Algorithm) -> ");
System.out.print(
CompletableFuture.supplyAsync(() -> fiboDP(position)) // Function Call...
.orTimeout(10, TimeUnit.SECONDS)
.join()
+ "\n"
);
}
catch (CompletionException exception) {
System.out.print(decodeException(exception) + "\n"); }

// Output for Space Optimized Algorithm...
try {
System.out.print("OUTPUT (Space Optimized Algorithm) -> ");
System.out.print(
CompletableFuture.supplyAsync(() -> fiboDPSO(position)) // Function Call...
.orTimeout(10, TimeUnit.SECONDS)
.join()
+ "\n"
);
}
catch (CompletionException exception) {
System.out.print(decodeException(exception) + "\n"); }

}

}

# Check for the correctness of the code. (Note- Do not include main function)
```

Looking at your Fibonacci implementation code, I can verify that the algorithms are correct. Here's my analysis:

Correctness Assessment:

1. Basic Recursive Method (`fiboAt(int position)`)

- ✓ Correct base cases (position 0 and 1)
- ✓ Proper recursive formula
- ✓ Input validation
- ⚠ Inefficient for large inputs (exponential time)

2. Memoization Method (`fiboAt(int position, long[] memory)`)

- ✓ Correct memoization implementation
- ✓ Proper null check and memory initialization
- ✓ Efficient reuse of computed values
- ✓ Maintains correct time complexity

3. Dynamic Programming Tabulation (`fiboDP(int position)`)

- ✓ Proper bottom-up approach
- ✓ Correct DP state definition
- ✓ Efficient iterative solution
- ✓ Proper base case handling

4. Space Optimized DP (`fiboDPSO(int position)`)

- ✓ Correct space optimization
- ✓ Proper variable management (`previousToPrevious`, `previous`, `current`)
- ✓ Maintains O(1) space complexity
- ✓ Correct iterative computation

Minor Issues Found:

1. In `fiboAt(int position, long[] memory)` :

```
java

// The comments say memory[0] and memory[1] initialization is "not required"
// but it's actually necessary for correctness
memory[0] = 0; // This IS required
memory[1] = 1; // This IS required
```

2. **Input validation** is consistent across all methods ✓
3. **Edge cases** are properly handled (position 0, position 1) ✓

Suggested Improvements:

```
java

// In the memoization method, consider this cleaner initialization:
public static long fiboAt(int position, long[] memory) {
    validatePosition(position);

    if (position == 0 || position == 1) return position;

    if (memory == null) {
        memory = new long[position + 1];
        Arrays.fill(memory, -1);
        memory[0] = 0;
        memory[1] = 1;
    }

    if (memory[position] == -1) {
        memory[position] = fiboAt(position - 1, memory) + fiboAt(position - 2, memory);
    }

    return memory[position];
}
```

Overall Assessment: The code is functionally correct and implements all four Fibonacci algorithms properly. The time and space complexity analyses in the comments are accurate.

Please check again whether the minor change is necessary?

You're absolutely right to question this! Let me re-examine the memoization method more carefully:

Re-analysis of the Memoization Method:

Looking at the current code:

```
java

if (memory == null) {
    memory = new long[position + 1];
    Arrays.fill(memory, Long.MIN_VALUE);
    memory[0] = 0; // "Not required"
    memory[1] = 1; // "Not required"
}
```

Actually, the current implementation is correct and the initializations ARE required. Let me trace through why:

Why the initializations are necessary:

1. For position = 2:

- fiboAt(2, null) calls fiboAt(1, memory) and fiboAt(0, memory)
- When fiboAt(1, memory) is called, it checks memory[1] which is Long.MIN_VALUE
- So it recursively computes fiboAt(0, memory) + fiboAt(-1, memory) ← ERROR!

2. Without the initializations, the recursion would continue to negative positions, causing infinite recursion or errors.

The current code correctly:

- ✓ Initializes base cases in memory
- ✓ Prevents unnecessary recursion for base cases
- ✓ Uses Long.MIN_VALUE as a clear "uncomputed" sentinel value

Correction to My Previous Assessment:

The minor change I suggested is NOT necessary. Your original implementation is correct. The comments saying "Not required" are actually misleading - those initializations ARE required for the algorithm to work correctly.

Final Verdict: Your memoization implementation is perfectly correct as written. No changes needed!