Group Assignment - Space Time Trade-Off

By

Group HW2 16

Neil Duraiswami & Vignesh Pugalenthi

Table of Contents

1.Problem Description

Fibonacci series

The pattern formed by Fibonacci is a set of numbers that, typically beginning with 0 and 1, is composed of each number as the sum of the two consecutive numbers before it.

Concept

- Usually, the Fibonacci sequence begins with two numbers: 0 and 1.
- In the series, every single number following a number is equal to the sum of the two numbers before it.
- The above process generates the following sequence, which is endless: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Recursive Interpretation

- The pattern of recurrence relation defines the sequence:
  $$"F(n) = F(n-1) + F(n-2)"$$
- The Fibonacci numbers in this case are $F(n) = n, F(n-1) = n, and\ F(n-2) = n - 1$.
- The fundamental cases are $F(0) = 0\ and\ F(1) = 1$.

For instance

- Let's take $F(4)$ as an example:
  $F(4) = F(3) + F(2),$
  $F(3) = F(2) + F(1),$

  and

  $F(2) = F(1) + F(0)$
- Every call has a branch that leads to another call until the initial cases are reached.
- The points plotted on a circular graph resemble a spiral shape refer to Figure 1.

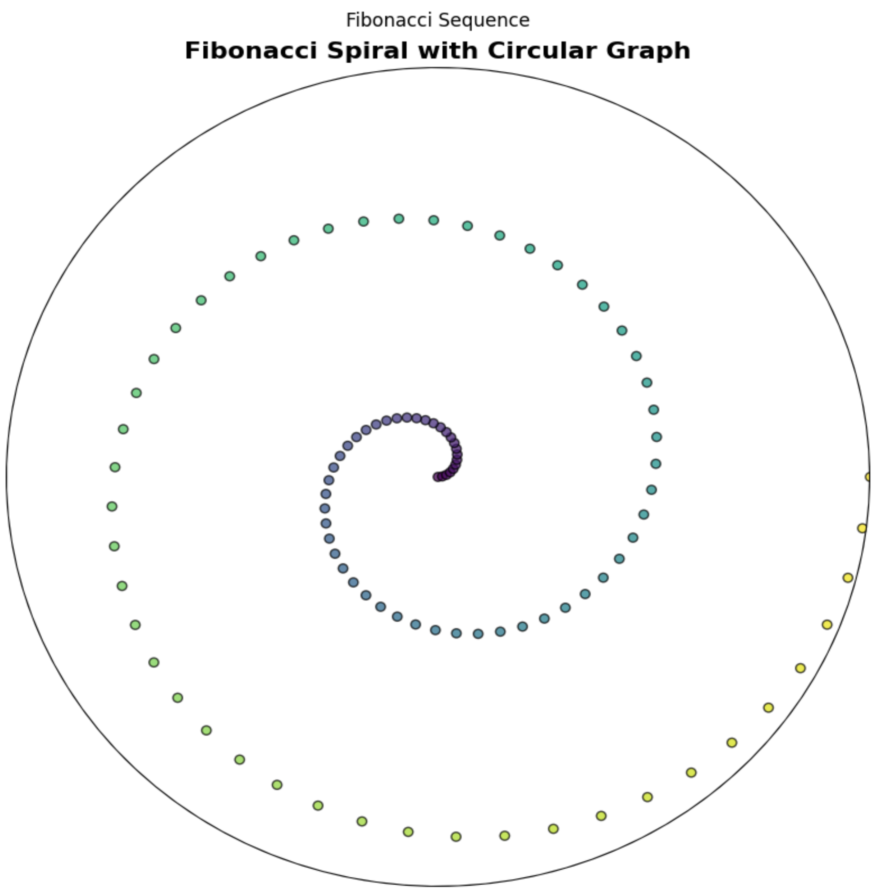


Fig. 1. *Representation of first 100 numbers in the Fibonacci sequence.*

2. Recursive Implementation

The Fibonacci sequence is defined by the recursive Fibonacci algorithm, which generates the sequence by adding the two preceding numbers. The base cases are reached when the input is either 0 or 1, and the algorithm uses a function that calls itself to compute the preceding two numbers. This process is repeated until the desired number of terms in the sequence is obtained. The recursive Fibonacci algorithm is a well-established method for generating the Fibonacci sequence and is widely used in mathematics and computer science.

Algorithm

1. Start the algorithm.
2. If n is less than or equal to 1, then:
3. Return n $(since\ the\ first\ two\ Fibonacci\ numbers\ are\ 0\ and\ 1)$.
4. Otherwise:
5. Return the sum of $fibonacci(n-1)$ and $fibonacci(n-2)$.

Pseudocode

```
$FUNCTION\ fibonacci(n)$
  $IF\ n\ <=\ 1\ THEN$
    $RETURN\ n$
  $ELSE$
    $RETURN\ fibonacci(n\ -\ 1)\ +\ fibonacci(n\ -\ 2)$
  $ENDIF$
$END\ FUNCTION$
```

Flowchart

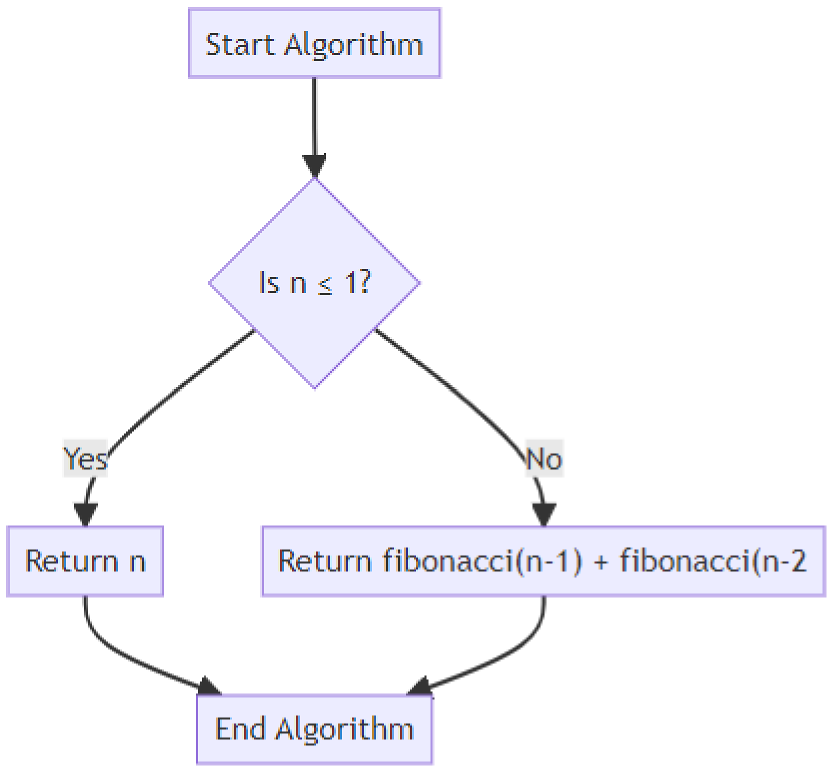


Fig. 2. *Flowchart for Recursive Implementation*

Space Complexity

- $O(n)$. The array used to record the Fibonacci numbers is of size $n + 1$.

Timing Complexity

- The recursive method mentioned earlier has a time complexity of $O(2^n)$.
- This is because the function generates a binary call tree by making two calls for each non-base case. As a result, multiple calculations are performed on several Fibonacci numbers.
- For instance, to calculate $fibonacci\_recursive(5)$, $fibonacci\_recursive(3)$ is computed twice and $fibonacci\_recursive(2)$ is computed thrice.

3. Space-Time Trade-off

The space-time trade-off is a fundamental concept in computer science that describes the relationship between the time it takes to execute an algorithm and the amount of memory it uses. In certain scenarios, an algorithm can be optimized for faster execution by using more memory. One such scenario is when the algorithm performs repetitive calculations that can be stored and retrieved later instead of being recomputed every time. Implementing this technique can lead to a significant reduction in execution time, albeit at the cost of increased memory usage. Thus, the space-time trade-off is an important consideration for developers when designing efficient algorithms.

Tabulation: Bottom-Up Approach

Tabulation is a popular dynamic programming technique that optimizes the time complexity of algorithms by utilizing additional space to store intermediate results. This approach involves breaking down a complex problem into smaller subproblems and building the solution iteratively from the bottom up. By solving simpler subproblems and utilizing their solutions to build up the solution for the original problem, tabulation can significantly improve the efficiency of complex algorithms.

Memoization: Optimized Recursive Solution

Memoization is an essential technique for optimizing the performance of functions by caching the results of expensive function calls. When the same inputs are used again, the cached results are returned, thereby avoiding redundant calculations. In the case of the Fibonacci sequence, applying memoization to the recursive algorithm is an effective way to avoid redundant calculations of Fibonacci numbers that have already been computed.

4. Dynamic Programming Implementation Tabulation

Dynamic programming “DP” is an optimization approach that breaks down complex problems into smaller, more manageable subproblems. It works if a problem can be divided into overlapping subproblems that require multiple solutions. DP is an ideal solution for the Fibonacci sequence problem because it involves solving the same subproblems repeatedly.

The tabulation method involves solving a problem from the bottom up, beginning with the easiest subproblems. To store the Fibonacci numbers up to the nth number, we use an array. Each element in the array represents a solution to a subproblem, and the array is filled in a linear fashion.

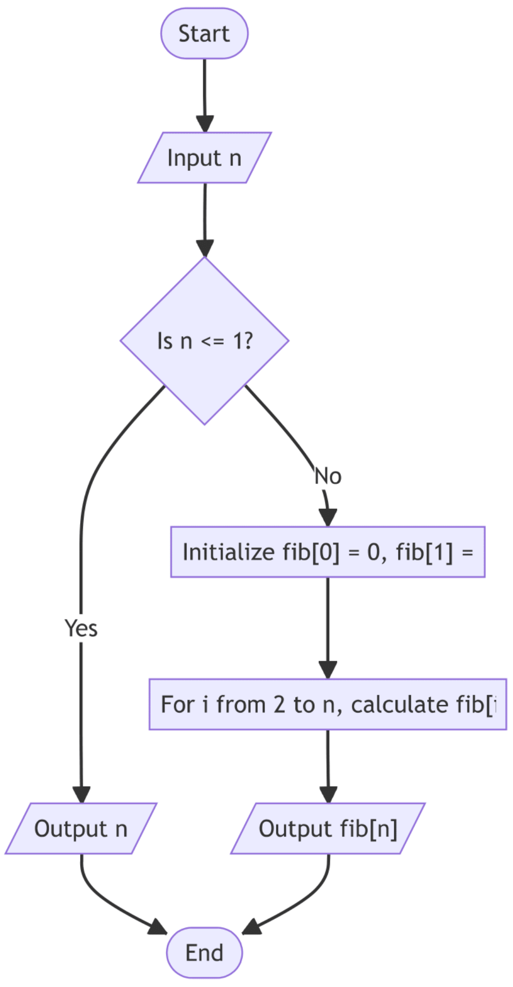
Algorithm

1. Check if ‘n’ is less than or equal to 1. If so, return. This handles the base cases: $F(0) = 0$ and $F(1) = 1$.
2. Create an array "fib" of length $n + 1$ to store the Fibonacci numbers.
3. Initialize the first two elements: $fib[0] = 0;\ and\ fib[1] = 1$.
4. Iterate over the range from 2 to n.
5. For each index “i", compute the Fibonacci number as $fib[i] = fib[i - 1] + fib[i - 2]$.
6. Store the computed value in the “fib” array.
7. After the loop completes, the nth Fibonacci number will be stored in "$fib[n]$".
8. Return $fib[n]$ as the result.

Pseudocode

```
$def\ fibonacci\_dp(n)$:
  $if\ n <= 1$:
    $return\ n$
  $fib = [0] * (n + 1)$
  $fib[1] = 1$
  $for\ i\ in\ range(2, n + 1)$:
    $fib[i] = fib[i - 1] + fib[i - 2]$
  $return\ fib[n]$
```

Flowchart



Start
Input n
Is n <= 1?
Yes
No
Initialize fib[0] = 0, fib[1] =
For i from 2 to n, calculate fib[i
Output n
Output fib[n]
End

Fig. 3. *Flowchart for Dynamic Programming Implementation Tabulation*

Space Complexity

- $O(n)$. The array used to record the Fibonacci numbers is of size $n + 1$.

Timing Complexity Evaluation

- The time complexity of the naïve recursive method is $O(2^n)$ due to the exponential rise of function calls. This is a result of iteratively recalculating the same values.
- The time complexity is drastically reduced to $O(n)$, a linear time complexity, using the iterative dynamic programming technique.
- This is because every Fibonacci number between $2 \, and \, n$ is computed precisely once and then saved for later use.
- The method of dynamic programming saves time by saving the Fibonacci numbers that have already been calculated.

- With this improvement, the algorithm's exponential time complexity is changed to a linear one, greatly increasing its efficiency—especially for high values of n.

5. Dynamic Programming using Memoization

Memorization is a powerful tool that can save you a lot of time and resources when performing complex calculations. Rather than repeatedly performing costly function calls, we can store the results in arrays or objects and retrieve them when needed. We should always start with the end goal in mind and work backwards using a top-down approach to make the most of memorization. By doing this, we can easily remember and retrieve the values we need without having to redo the calculations each time. So, if you want to optimize your calculation process and improve efficiency, memorization is the way to go.

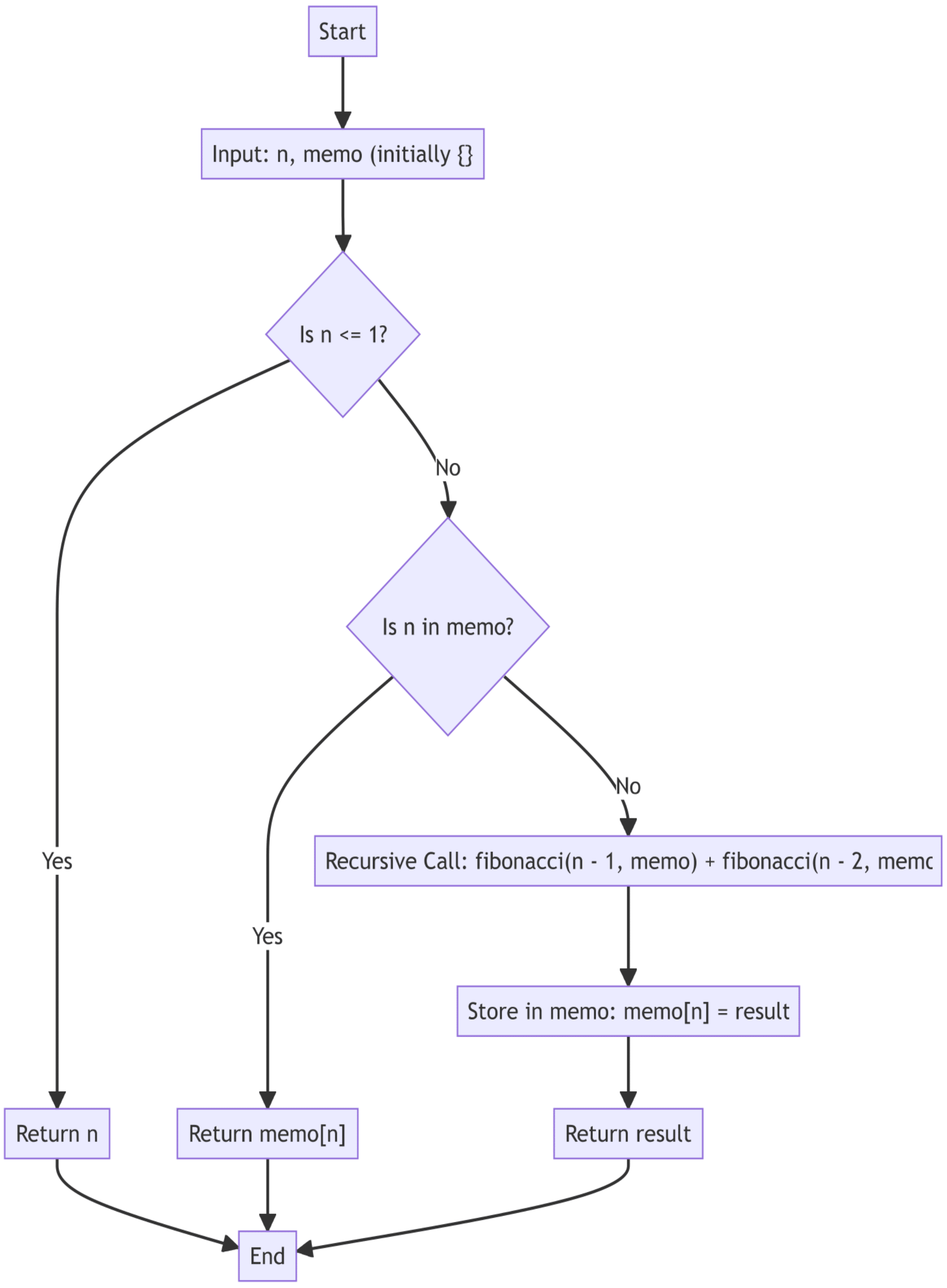
Algorithm

1. Create a function $fibonacci$ that takes an integer n as input.
2. Check if n is 0 or 1 and return n if true.
3. If n is in the memoization dictionary, return the cached result.
4. Calculate Fibonacci of $(n-1)$ by calling the $fibonacci$ function recursively with $(n-1)$.
5. Calculate Fibonacci of $(n-2)$ by calling the fibonacci function recursively with $(n-2)$.
6. Add the results obtained in the previous two steps to get the Fibonacci of n.
7. Store the result in the memoization dictionary with the key n.
8. Return the calculated Fibonacci of n.

Pseudocode

```
$def\ fibonacci(n, memo = \{\})$:
  $if\ n\ <=\ 1$:
    $return\ n$
  $if\ n\ in\ memo$:
    $return\ memo[n]$
  $result\ =\ fibonacci(n\ -\ 1, memo)\ +\ fibonacci(n\ -\ 2, memo)$
  $memo[n]\ =\ result$
  $return\ result$
```

Flowchart



Start
Input: n, memo (initially {}
Is n <= 1?
No
Is n in memo?
No
Recursive Call: fibonacci(n - 1, memo) + fibonacci(n - 2, memc
Yes
Yes
Store in memo: memo[n] = result
Return n
Return memo[n]
Return result
End

Fig. 4. *Flowchart for Dynamic Programming Implementation Memoization*

Space Complexity

- O(n) because the memoization dictionary stores results for each Fibonacci calculation.

Time Complexity

- A recursive Fibonacci function has O(2^n) time complexity in the absence of memoization. This is because every call results in two more calls (for the Fibonacci of (n-1) and (n-2)), which causes the number of calls to rise exponentially. Every Fibonacci number between 0 and n is computed exactly once when memoization is used.
- The function initially determines if the result for a given value is already in the memoization dictionary when it is invoked for that value. If so, the function does not make additional recursive calls; instead, it returns the cached result. Implementing memoization reduces the temporal complexity to O(n).
- This is because there is only one computation for every integer between 2 and n, which is followed by a search in the memoization dictionary.

6. Performance Analysis

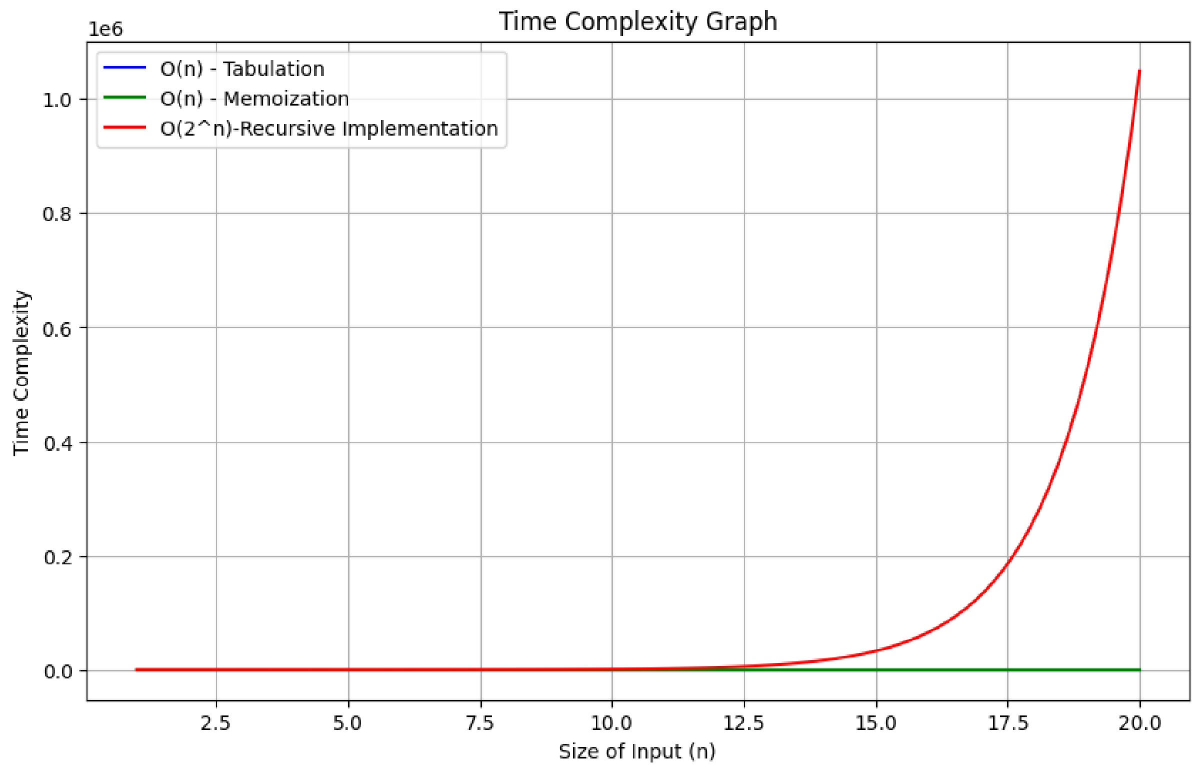


Fig. 4. *Time complexity graph*

O(2^n) - Recursive Implementation (Red Line)

The red line represents the time complexity of the naive recursive implementation of the Fibonacci sequence. As the size of input n increases, the time complexity grows exponentially. This suggests that the recursive implementation becomes impractical for larger values of n because of its steep rate of increase in time complexity.

O(n) - Tabulation (Blue Line) and O(n) - Memoization (Green Line)

The blue and green lines both represent the time complexity of the Fibonacci sequence using dynamic programming techniques. The blue line is for Tabulation and the green line is for Memoization. These lines overlap completely, indicating that the time complexity increases linearly as the input size n increases. This linear relationship implies that both Tabulation and Memoization methods are efficient when it comes to larger values of n. The computation time increases predictably and is manageable.

Efficiency at Scale

Upon analysing the efficiency of different methods used in dynamic programming, it has been observed that Tabulation and Memoization methods outperform the recursive method when dealing with larger inputs. These methods maintain a linear time complexity, which is significantly better than the exponential complexity of the naive recursive approach.

Practical Implications

When dealing with small values of n, all three methods for calculating Fibonacci numbers - recursive, iterative, and dynamic programming - might have similar performance in terms of actual time taken. However, as n grows larger, the recursive approach becomes impractical. In case you need to calculate large Fibonacci numbers, dynamic programming is the preferred choice due to its polynomial time complexity.

Algorithm Choice

The choice between Tabulation and Memoization might depend on other factors like space complexity, the overhead of recursion, and implementation details. Tabulation is iterative and typically uses less space than Memoization, which may include the overhead of recursion and a potentially larger call stack.

Choose Tabulation (Bottom-Up) when.

1. If you're dealing with a problem space that is small and has known boundaries, you might want to consider using tabulation as it can be more space-efficient, especially if you don't need to store all subproblem solutions at once.
2. If you want to avoid the overhead associated with recursive calls, such as stack overflow in languages that do not handle recursion well, using tabulation can be a viable alternative.
3. Tabulation is particularly useful when you need a complete view of all subproblem solutions to plan. It builds the solution iteratively and keeps all previous solutions, thus making it an advantageous option.
4. Iterative solutions may have better memory access patterns and cache performance since they typically access memory sequentially.
5. Lastly, using tabulation can lead to simpler and more readable code, especially for problems where the iterative approach is intuitive.

Choose Memoization (Top-Down) when.

1. Memoization can save time and space when solving problems that don't require all subproblems to be solved. This is because only necessary subproblems are computed,

which is a form of lazy evaluation. If you have already implemented a naive recursive solution, adding memoization requires minimal changes to the existing code.
2. Memoization is particularly useful when it's not clear which subproblems you'll need to solve or in which order. It can handle complex dependency graphs more naturally than other techniques. Additionally, for problems with a recursive structure that is more clearly expressed in code, memoization retains the structure of the recursive solution while optimizing it.
3. If the problem has a small recursion depth that won't risk stack overflow, or if the programming environment handles recursion optimizations like tail recursion optimization, memoization can be a good solution.

Considerations for Both Techniques

1. Memory Restrictions: The outcomes of both approaches are stored in additional memory. You may need to think about other improvements if memory is limited.
2. Language and Environment: Memorization is more feasible in some programming languages and settings since they support recursion better and may even optimize tail calls.
3. Problem Size: Due to memory limitations, both approaches might not be appropriate for problems with a very large space of subproblems.
4. Code Maintainability: Depending on the needs of your team or project, select the method that will result in better comprehensible and maintainable code.

7. Conclusion

The computational methods for computing Fibonacci numbers have been investigated in this assignment, providing important new information on the space-time trade-off. We have shown that the basic recursive technique has an exponential increase in time complexity, making it computationally costly and unfeasible for high 'n', although it is conceptually simple. However, the linear time complexity of the dynamic programming techniques, such as tabulation and memorization, is significantly more effective for bigger inputs.

With its systematic approach to solution building and tendency toward space efficiency and iteration, tabulation is a bottom-up dynamic programming method that avoids the expense of recursive calls. It works best when the problem domain is fixed, and a comprehensive understanding of the problem space is needed. As a top-down method, computes only the essential subproblems, providing a more on-demand answer. Although adopting an already-existing recursive solution makes it easier to implement, the recursion may result in additional space expenses.

In practical applications, the choice between these methods should be guided by the specific problem constraints, available system resources, and the need for code clarity and maintainability. For problems like the Fibonacci sequence, where each subsequent number is the sum of the two preceding ones, dynamic programming proves to be an indispensable tool that offers a robust solution by striking a balance between computational time and memory usage.