



DYNAMIC PROGRAMMING

Analysis of Algorithms

Semester Project Report

○ Table of Contents

○ <i>Dynamic Programming</i>	3
------------------------------------	---

○ <i>Top-Down Approach</i>	4
----------------------------------	---

○ <i>Bottom-up Approach</i>	5
-----------------------------------	---

○ <i>Python Code for Recursive and DP implementation on Fibonacci Sequence with Recursion</i>	6
---	---

○ <i>Output & Comparison Table</i>	7
--	---

○ <i>Comparison Chart & Time & space Complexity</i>	8
---	---

DYNAMIC PROGRAMMING

Dynamic programming is a powerful algorithmic technique that allows us to solve complex problems by breaking them down into simpler, overlapping subproblems. It is particularly useful when we encounter repetitive calculations or overlapping subproblems in our problem-solving process. The main idea behind dynamic programming is to solve each subproblem only once and store its solution for future use. This approach avoids redundant calculations and leads to significant performance improvements, especially for problems with exponential time complexity.

There are two key attributes that a problem must have for dynamic programming to be applicable **optimal substructure** and **overlapping sub-problems**. If a problem can be solved by combining optimal solutions to non-overlapping sub-problems, the strategy is called "**divide and conquer**" instead. This is why merge sort and quick sort are not classified as dynamic programming problems.

1. Optimal substructure

Optimal substructure means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its sub-problems. Such optimal substructures are usually described by means of recursion.

2. Overlapping sub-problems

Overlapping sub-problems means that the space of sub-problems must be small, i.e., any recursive algorithm solving the problem should solve the same sub-problems over and over, rather than generating new sub-problems.

For example:

Consider the recursive formulation for generating the Fibonacci series: $F_i = F_{i-1} + F_{i-2}$, with base case $F_1 = F_2 = 1$. Then $F_3 = F_2 + F_1$, and $F_4 = F_3 + F_2$. Now F_3 is being solved in the recursive sub-trees of both F_4 as well as F_5 . Even though the total number of sub-problems is small (only 43 of them), we end up solving the same problems over and over if we adopt a naive recursive solution such as this. Dynamic programming takes account of this fact and solves each sub-problem only once.

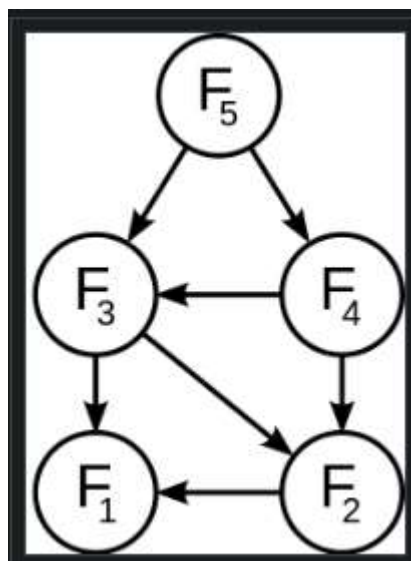


Figure 1

This can be achieved in either of two ways.

- Top-Down Approach
- Bottom-up Approach

1. Top-Down Approach

This is the direct fall-out of the recursive formulation of any problem. If the solution to any problem, can be formulated recursively using the solution to its sub-problems, and if its sub-problems are overlapping, then one can easily memoize or store the solutions to the sub-problems in a table. Whenever we attempt to solve a new sub-problem, we first check the table to see if it is already solved. If a solution has been recorded, we can use it directly, otherwise we solve the sub-problem and add its solution to the table.

Example:

Using dynamic programming in the calculation of the nth member of the Fibonacci sequence improves its performance greatly. Here is a naïve implementation.

```
function fib(n)
  if n <= 1 return n
  return fib(n - 1) + fib(n - 2)
```

Figure 2

Notice that if we call, say, fib (5), we produce a call tree that calls the function on the same value many different times:

```
1. fib(5)
2. fib(4) + fib(3)
3. (fib(3) + fib(2)) + (fib(2) + fib(1))
4. ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
5. (((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
```

Figure 3

In particular, fib (2) was calculated three times from scratch. In larger examples, many more values of fib, or subproblems are recalculated, leading to an exponential time algorithm.

Suppose we have a simple map object, m, which maps each value of fib that has already been calculated to its result, and we modify our function to use it and update it. The resulting function requires only O(n) time instead of exponential time but requires O(n) space.

```

# Initialize a map with base cases for Fibonacci numbers
m = {0: 0, 1: 1}

# Recursive function to calculate Fibonacci number using memoization
2 usages
def fib(n):
    # Check if the Fibonacci number is already computed
    if n not in m:
        # Compute the Fibonacci number recursively and store it in the map
        m[n] = fib(n - 1) + fib(n - 2)
    # Return the Fibonacci number
    return m[n]

```

Figure 4

This technique of saving values that have already been calculated is called **memorization**. This is the **top-down approach**. Since we first break the problem into subproblems and then calculate and store values.

2. Bottom-up Approach

Once we formulate the solution to a problem recursively as in terms of its sub-problems, we can try reformulating the problem in a bottom-up fashion: try solving the sub-problems first and use their solutions to build-on and arrive at solutions to bigger sub-problems. This is also usually done in a tabular form by iteratively generating solutions to bigger and bigger sub-problems by using the solutions to small sub-problems.

Example:

In the bottom-up approach, we calculate the smaller values of fib first, then build larger values from them. This method also uses $O(n)$ time since it contains a loop that repeats $(n - 1)$ times, but it only takes constant ($O(1)$) space, in contrast to the top-down approach which requires $O(n)$ space to store the map.

```

# Function to calculate Fibonacci number iteratively
def fib(n):
    if n == 0:
        return 0
    else:
        previousFib = 0
        currentFib = 1
        for _ in range(n - 1): # Repeat n - 1 times
            newFib = previousFib + currentFib
            previousFib = currentFib
            currentFib = newFib
        return currentFib

```

Figure 5

In this example, we have only calculated fib(2) one time, and then use it to calculate both fib(4) and fib(3), instead of computing it every time either of them is evaluated.

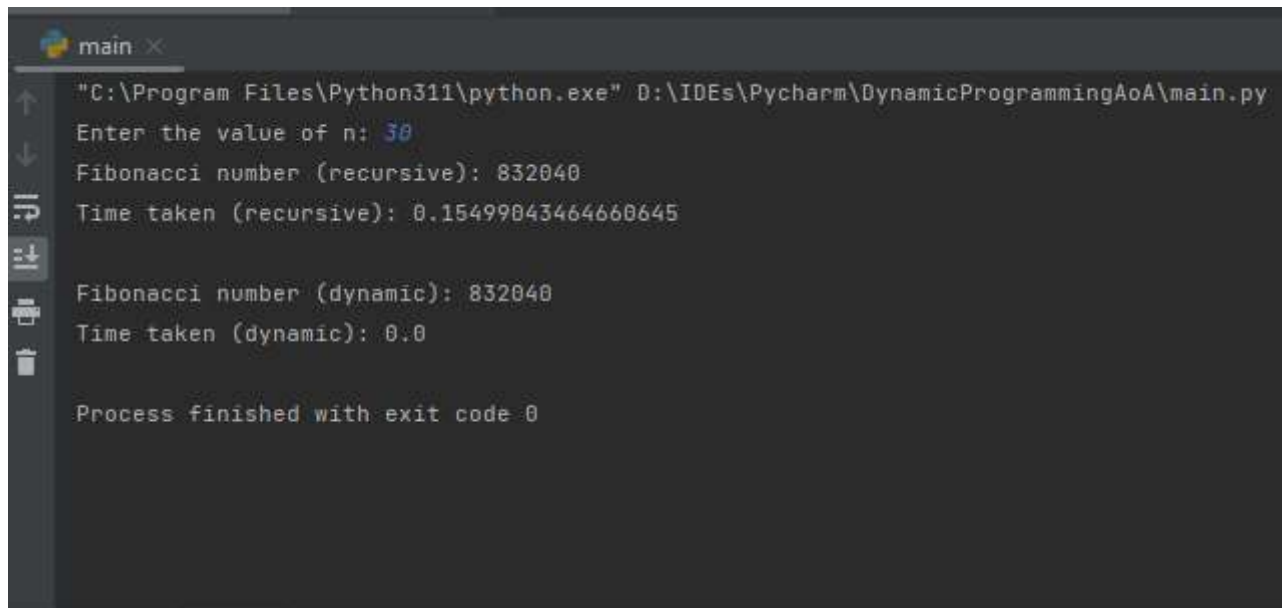
Python Code for Recursive and DP implementation on Fibonacci Sequence with Recursion

CODE:

```
main.py x
1 import time
2 # Recursive implementation of Fibonacci sequence
3 usages
4 def fibonacci_recursive(n):
5     if n <= 1:
6         return n
7     return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)
8
9 # Dynamic programming implementation of Fibonacci sequence with recursion
10 usages
11 def fibonacci_dynamic(n, m={}):
12     if n in m:
13         return m[n]
14     if n <= 1:
15         result = n
16     else:
17         result = fibonacci_dynamic(n - 1, m) + fibonacci_dynamic(n - 2, m)
18     m[n] = result
19     return result
20
21 # Time comparison
22 n = int(input("Enter the value of n: "))
23
24 start_time = time.time()
25 fib_recursive = fibonacci_recursive(n)
26 end_time = time.time()
27 recursive_time = end_time - start_time
28
29 start_time = time.time()
30 fib_dynamic = fibonacci_dynamic(n)
31 end_time = time.time()
32 dynamic_time = end_time - start_time
33
34 print("Fibonacci number (recursive):", fib_recursive)
35 print("Time taken (recursive):", recursive_time)
36 print("\nFibonacci number (dynamic):", fib_dynamic)
37 print("Time taken (dynamic):", dynamic_time)
```

Figure 6

Output:



```
main X
"C:\Program Files\Python311\python.exe" D:\IDEs\Pycharm\DynamicProgrammingAoA\main.py
Enter the value of n: 30
Fibonacci number (recursive): 832040
Time taken (recursive): 0.15499843464660645

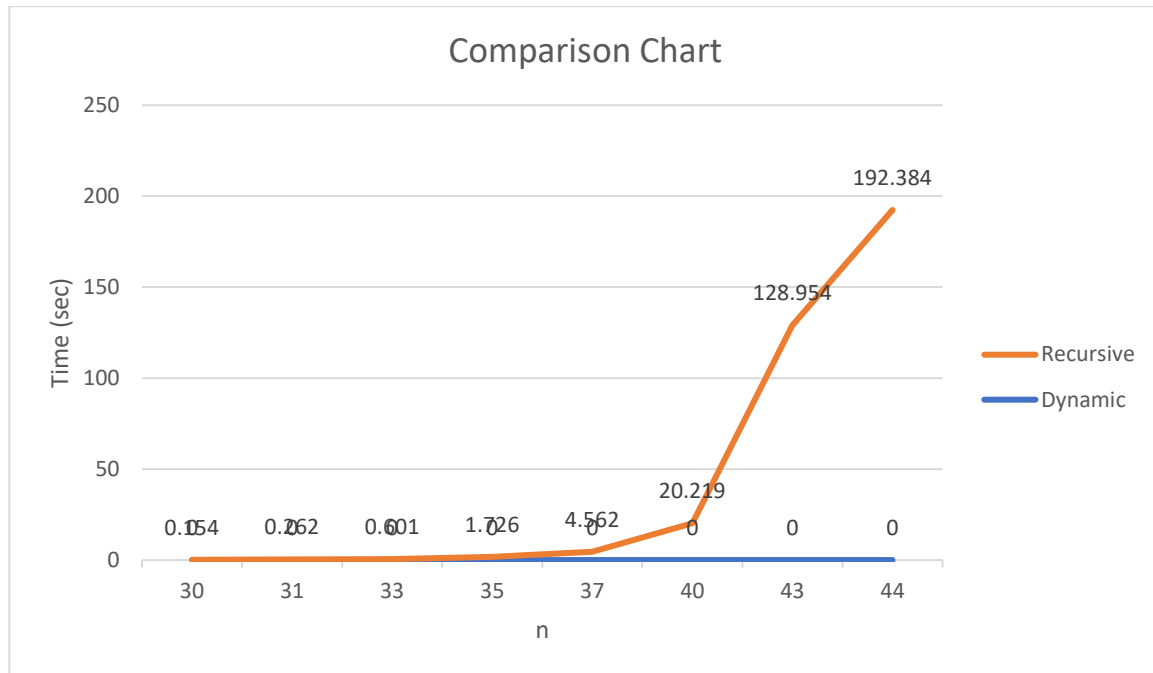
Fibonacci number (dynamic): 832040
Time taken (dynamic): 0.0

Process finished with exit code 0
```

Figure 7

Comparison Table

n	Recursive Fibonacci	Dynamic Fibonacci	Fibonacci No
30	0.154	0.0	832040
31	0.262	0.0	1346269
33	0.601	0.0	3524578
35	1.726	0.0	9227465
37	4.562	0.0	24157817
40	20.219	0.0	102334155
43	128.954	0.0	433494437
44	192.384	0.0	701408733



Time & Space Complexity

The complexity of the provided code can be analyzed as,

1. Recursive Implementation

- Time Complexity: The time complexity of the recursive Fibonacci implementation without memoization is exponential. It can be roughly approximated as $O(2^n)$, where n is the input value. This is because for each Fibonacci number, the function makes two recursive calls, leading to an exponential growth in the number of function calls and computations.
- Space Complexity: The space complexity of the recursive implementation without memoization is $O(n)$, where n is the depth of the recursion. This is because the function calls are stacked on the call stack, and the stack depth corresponds to the input value n .

2. Dynamic Programming Implementation (with Memoization)

- Time Complexity: The time complexity of the dynamic programming implementation with memoization is significantly improved compared to the recursive implementation. The memoization table allows for storing and reusing previously computed Fibonacci numbers, avoiding redundant calculations. As a result, the time complexity becomes linear or $O(n)$, where n is the input value. This is because each Fibonacci number is computed once and stored in the memoization table, and subsequent lookups take constant time.
- Space Complexity: The space complexity of the dynamic programming implementation with memoization is $O(n)$. This is because the memoization table stores the computed Fibonacci numbers up to n , occupying space proportional to the input value.