Problem Solving Strategies and Computational Approaches SCS1304

Handout 6 : Dynamic Programming

Prof Prasad Wimalaratne PhD(Salford),SMIEEE

Overview

- Introduction to Dynamic Programming (DP)
- Divide Conquer vs Dynamic Programming
- How Does DP Work?
- Examples of DP
- When to DP?
- Approaches to DP
  - Tabulation
  - Memoization

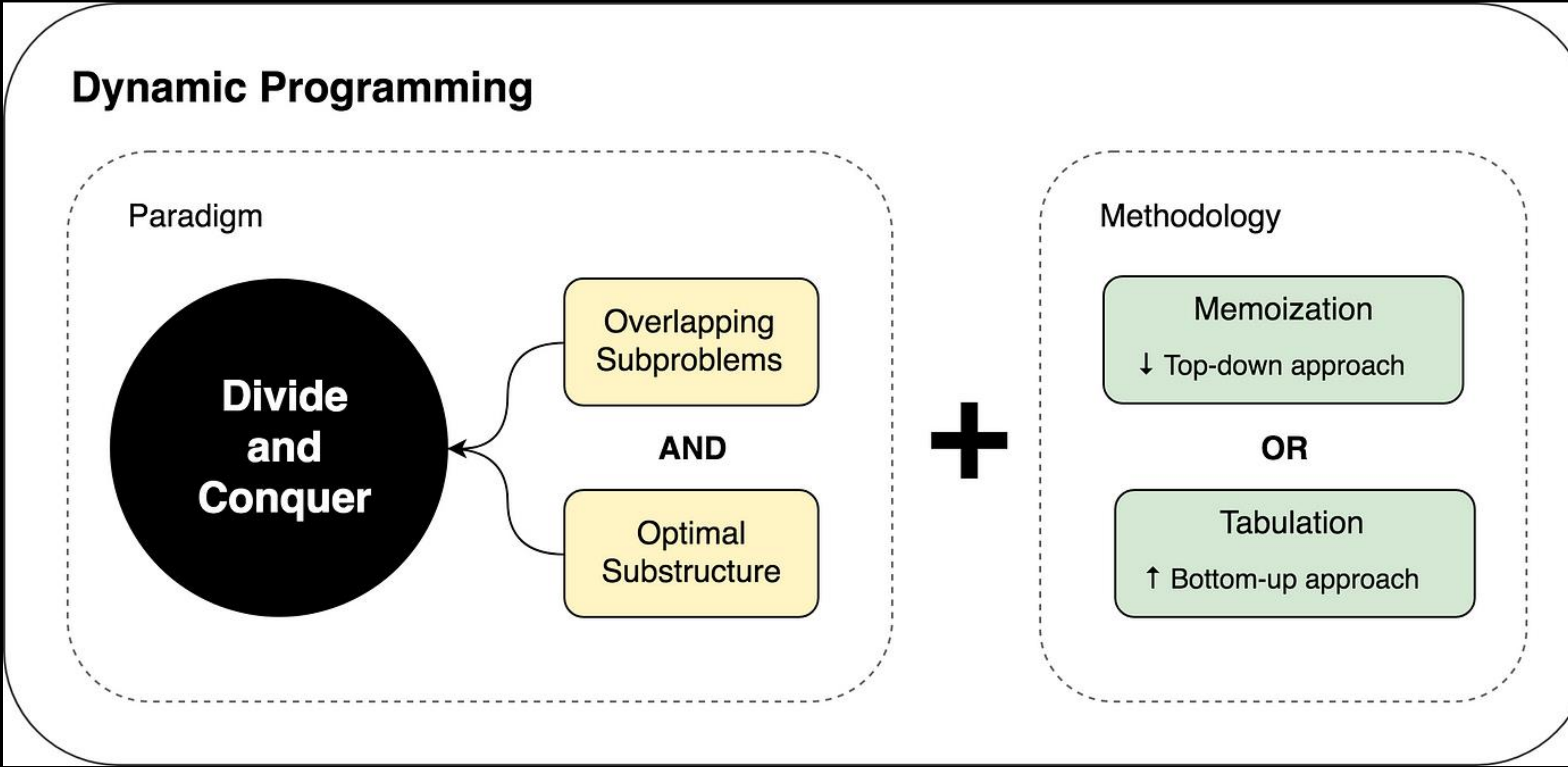
- Ref: 4 Principle of Optimality - Dynamic Programming introduction
  https://www.youtube.com/watch?v=5dRGRueKU3M

DP?: Real-Life Example: The Jigsaw Puzzle Analogy

- Dynamic programming can be understood with the analogy of solving a jigsaw puzzle efficiently. Imagine you are solving a large jigsaw puzzle. Instead of trying random pieces to see what fits (brute force), you use a systematic approach:

1. Break It Down: You group the puzzle pieces by color or pattern (e.g., edges, sky, trees, etc.), making it easier to focus on smaller parts of the puzzle.
2. Reuse Your Work: Once you solve a smaller section (e.g., the sky), you do not revisit it or redo it. Instead, you place it aside, knowing it is solved. This is like memoization in dynamic programming.
3. Build Step by Step: After solving smaller sections, you combine them to form larger sections (e.g., attaching the sky to the trees). This is similar to tabulation, where you solve subproblems iteratively and build up to the final solution.
4. Avoid Redundancy: If you have already figured out where a piece fits, you do not test it again in other places. This prevents repetitive work and saves time.

https://unstop.com/blog/dynamic-programming



https://itnext.io/dynamic-programming-vs-divide-and-conquer-2fea680becbe

Dynamic programming

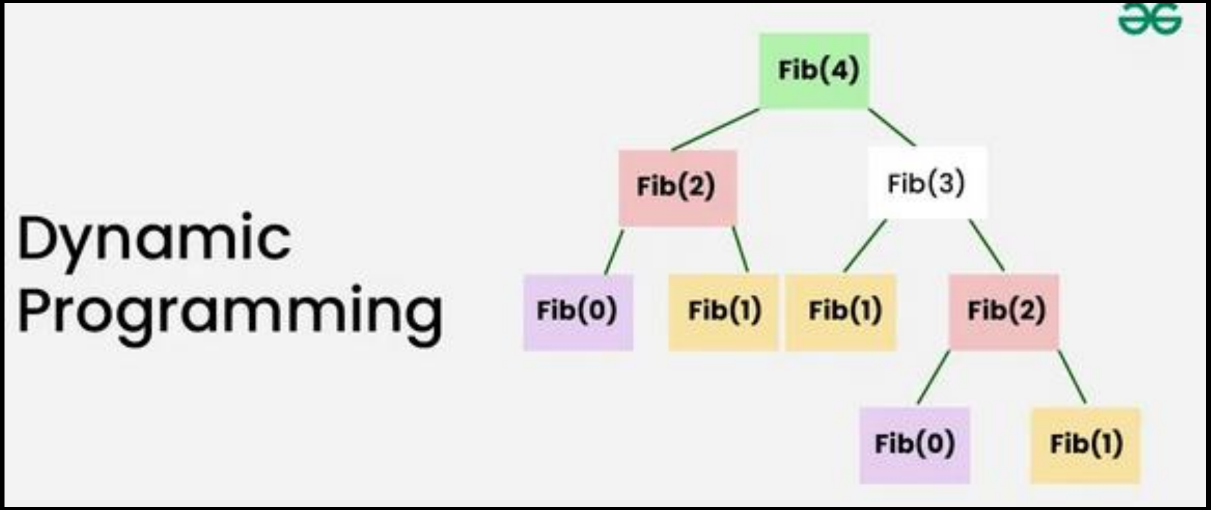
- Dynamic programming is an algorithmic technique used to solve complex problems by breaking them down into simpler, overlapping subproblems.
- The core idea is to avoid redundant computations by storing the solutions to these subproblems and reusing them when needed.
- This approach is particularly effective for optimization problems, where the goal is to find the best possible solution (e.g., minimum cost, maximum value).
- A technique where an algorithmic problem is first broken down into sub-problems, the results are saved, and then the sub-problems are optimized to find the overall solution which usually has to do with finding the maximum and minimum range of the algorithmic query.
- The presence of overlapping subproblems is a key characteristic that signals the applicability and effectiveness of dynamic programming.
- Richard Bellman was the one who came up with the idea for dynamic programming in the 1950s.

Dynamic programming

- Recursion? - the solution to a problem depends on solutions to its smaller subproblems.
- Meanwhile, dynamic programming is an optimization technique for recursive solutions.
- It is the preferred technique for solving recursive functions that make repeated calls to the same inputs.
- However, NOT all problems that use recursion can be solved by dynamic programming.
- **Unless solutions to the subproblems overlap, a recursion solution can only be arrived at using a divide-and-conquer method.
  - For example, problems like merge, sort, and quick sort are **NOT** considered dynamic programming problems.
  - This is because they involve putting together the best answers to subproblems that don't overlap.

Dynamic Programming

- Dynamic Programming is a method used in mathematics and computer science to solve complex problems by breaking them down into simpler subproblems.
- By solving each subproblem only once and storing the results, it avoids redundant computations, leading to more efficient solutions for a wide range of problems.
- Note the same colors in diagram below



https://www.geeksforgeeks.org/dynamic-programming/

Key Difference Between Divide and Conquer and Dynamic Programming

- Approach:
  - Divide and Conquer: Breaks a problem into smaller subproblems and solves them independently.
  - Dynamic Programming: Breaks a problem into overlapping subproblems and solves them recursively OR iteratively, storing their solutions.
- Subproblem Dependency:
  - Divide and Conquer: Subproblems are independent of each other.
  - Dynamic Programming: Subproblems overlap or share common subproblems.
- Solution Utilization:
  - Divide and Conquer: Does not utilize previously solved subproblem solutions.
  - Dynamic Programming: Utilizes previously solved subproblem solutions by storing them for reuse.
- Solution Construction:
  - Divide and Conquer: The solution is obtained by combining the solutions of independent subproblems.
  - Dynamic Programming: The solution is built by solving and combining overlapping subproblems iteratively OR recursively.

https://testbook.com/key-differences/difference-between-divide-and-conquer-and-dynamic-programming

Key Difference Between Divide and Conquer and Dynamic Programming ctd..

- Time Complexity:
  - Divide and Conquer: May take longer to solve the problem.
  - Dynamic Programming: Typically takes a shorter time to solve the problem.
- Subproblem Nature:
  - Divide and Conquer: Works well when subproblems are independent.
  - Dynamic Programming: Works well when subproblems overlap or share common subproblems.
- Example Algorithms:
  - Divide and Conquer: Merge Sort, Quick Sort.
  - Dynamic Programming: Fibonacci series, Knapsack problem.
- Memory Usage:
  - Divide and Conquer: Can be more memory-efficient in certain cases.
  - Dynamic Programming: May require more memory due to the storage of subproblem solutions.
- Approach Complexity:
  - Divide and Conquer: Generally simpler to understand and implement.
  - Dynamic Programming: Can be more complex to understand and implement.

https://testbook.com/key-differences/difference-between-divide-and-conquer-and-dynamic-programming

Divide and Conquer	Dynamic Programming
Breaks problems into smaller subproblems	Breaks problems into overlapping subproblems
Solves subproblems independently	Reuses solutions of overlapping subproblems
No shared information between subproblems	Stores solutions typically for future reference
Often results in redundant computations	Avoids redundant computations through memoization
Generally has a higher time complexity	Generally has a lower time complexity
Can be more intuitive for certain problems	Can be more efficient for problems with overlapping subproblems
Examples: Merge sort, Quick sort	Examples: Fibonacci sequence, Shortest path problems
May not exploit optimal substructure	Exploits optimal substructure to find optimal solutions

https://testbook.com/key-differences/difference-between-divide-and-conquer-and-dynamic-programming

Divide and Conquer	Dynamic Programming
Suitable for problems that can be divided into independent parts	Suitable for problems that can be solved by breaking them into overlapping subproblems
Examples: Binary search, Maximum subarray problem	Examples: Longest common subsequence, Knapsack problem
Requires solving all subproblems	Requires solving only necessary subproblems
No explicit storage of solutions	Usually Solutions are explicitly stored in a table or array
Examples: Binary exponentiation, Closest pair problem	Examples: Matrix chain multiplication, Longest increasing subsequence
Generally used when the subproblems are independent	used when the subproblems overlap
Examples: Mergesort, Binary search tree operations	Examples: Fibonacci series, Travelling salesman problem
May require a more detailed analysis of subproblems	Requires careful identification of overlapping subproblems
Breaks the problem into smaller, non-overlapping parts	Breaks the problem into overlapping subproblems
Intermedia results of subproblems are not store for future use	Intermedia results of subproblems are store for future use

https://testbook.com/key-differences/difference-between-divide-and-conquer-and-dynamic-programming

How Does Dynamic Programming (DP) Work?

Identify Subproblems: Divide the main problem into smaller, independent subproblems.

i. Store Solutions: Solve each subproblem and store the solution in a table or array.
ii. Build Up Solutions: Use the stored solutions to build up the solution to the main problem.
iii. Avoid Redundancy: By storing solutions, DP ensures that each subproblem is solved only once, reducing computation time.

https://www.geeksforgeeks.org/dynamic-programming/

Examples of Dynamic Programming (DP)

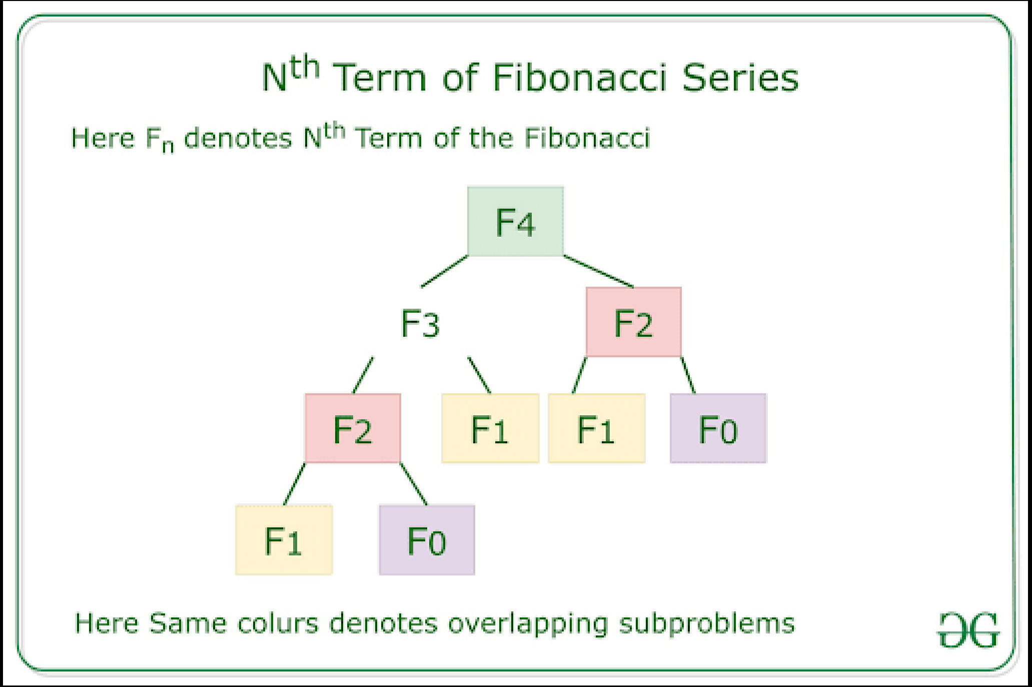
- Example 1: Consider the problem of finding the Fibonacci sequence:

  Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

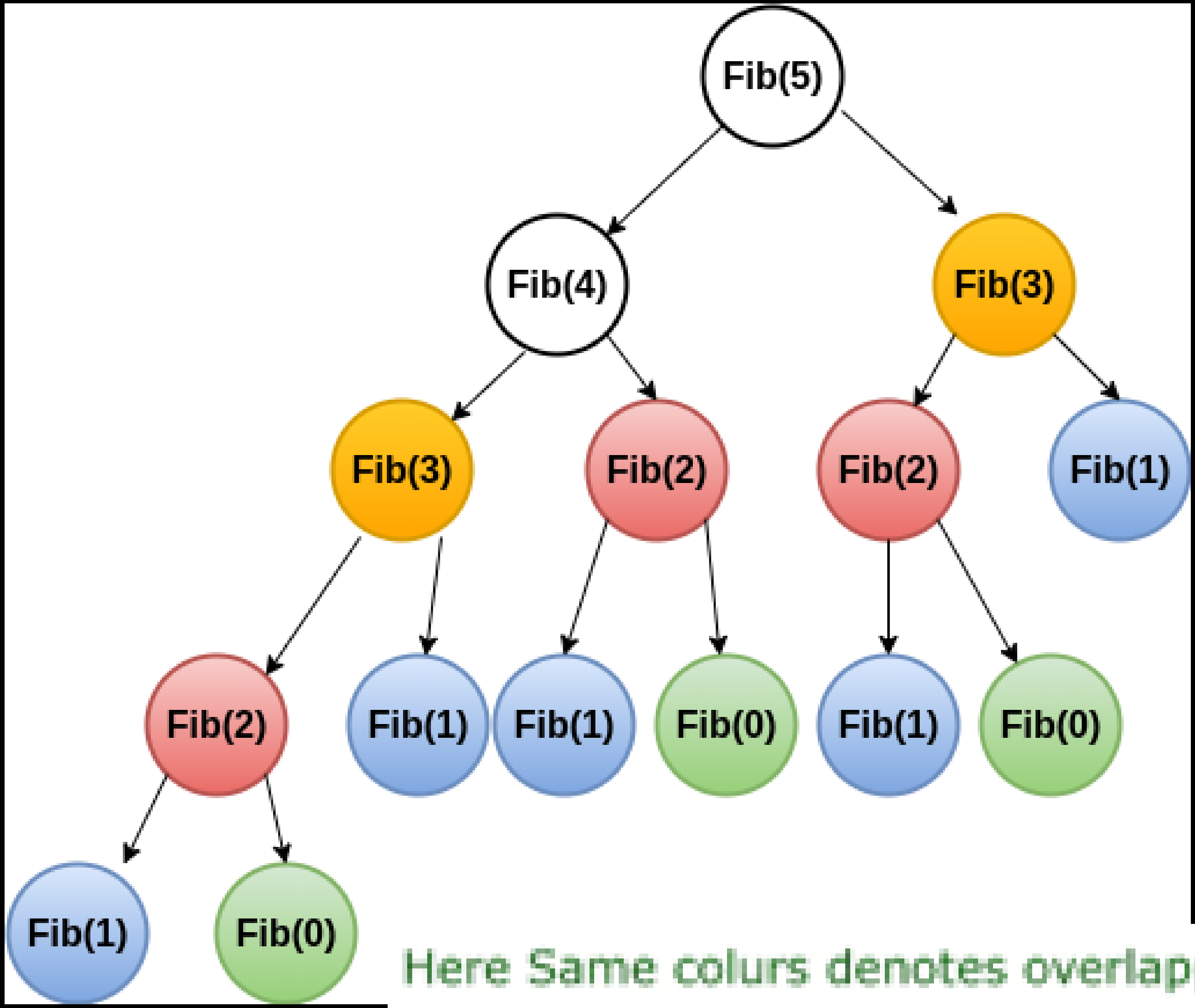
- Brute Force Approach: To find the nth Fibonacci number using a brute force approach, you would simply add the (n-1)th and (n-2)th Fibonacci numbers.
- This would work, but it would be inefficient for large values of n, as it would require calculating all the previous Fibonacci numbers.

Fibonacci Series using Dynamic Programming

- Subproblems: F(0), F(1), F(2), F(3), …
- Store Solutions: Create a table to store the values of F(n) as they are calculated.
- Build Up Solutions: For F(n), look up F(n-1) and F(n-2) in the table and add them.
- Avoid Redundancy: The table ensures that each subproblem (e.g., F(2)) is solved only once.

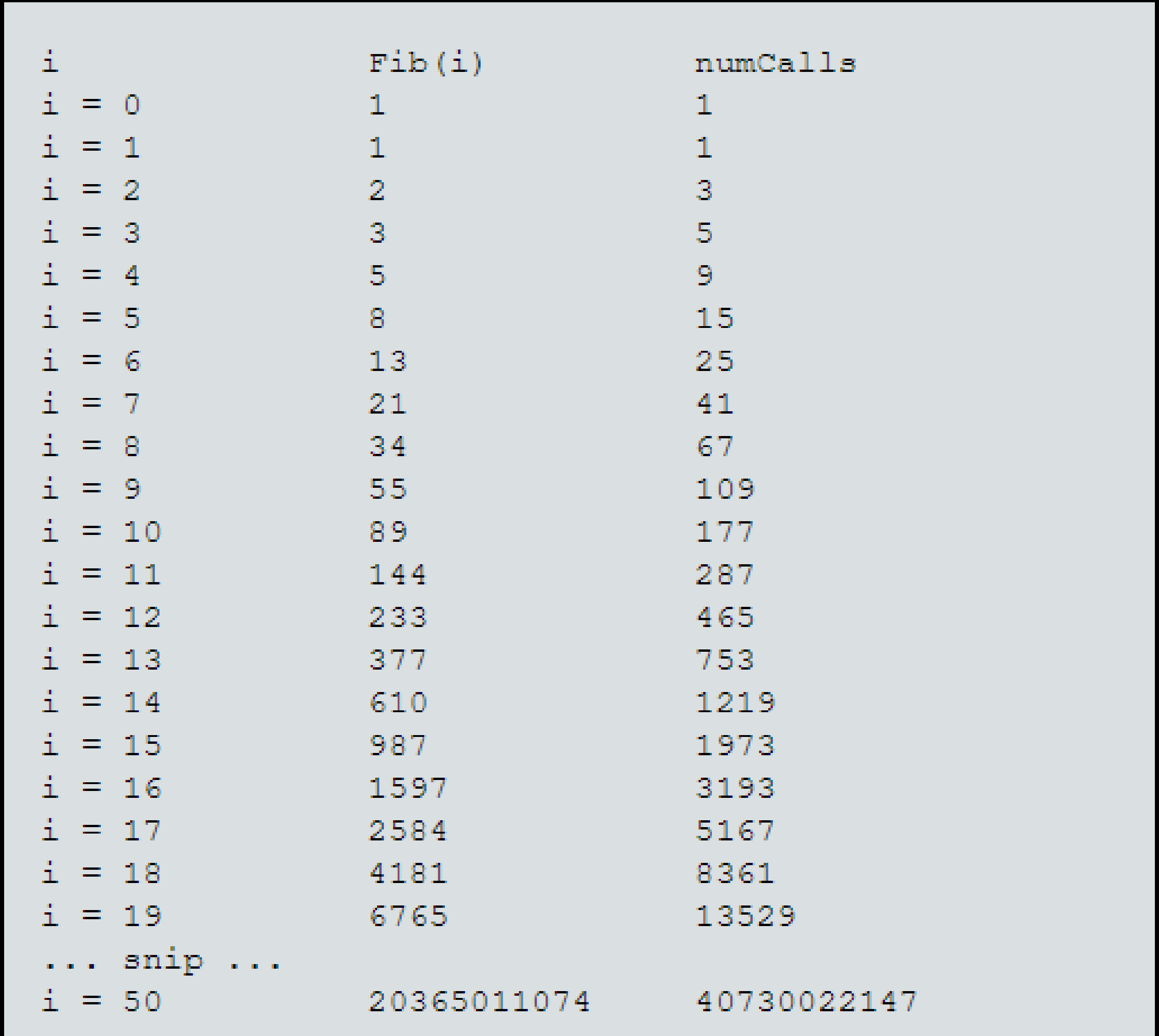

Nth Term of Fibonacci Series
Here Fn denotes Nth Term of the Fibonacci
F4
F3
F2
F2
F1
F1
F0
F1
F0
Here Same colurs denotes overlapping subproblems

https://www.geeksforgeeks.org/dynamic-programming/



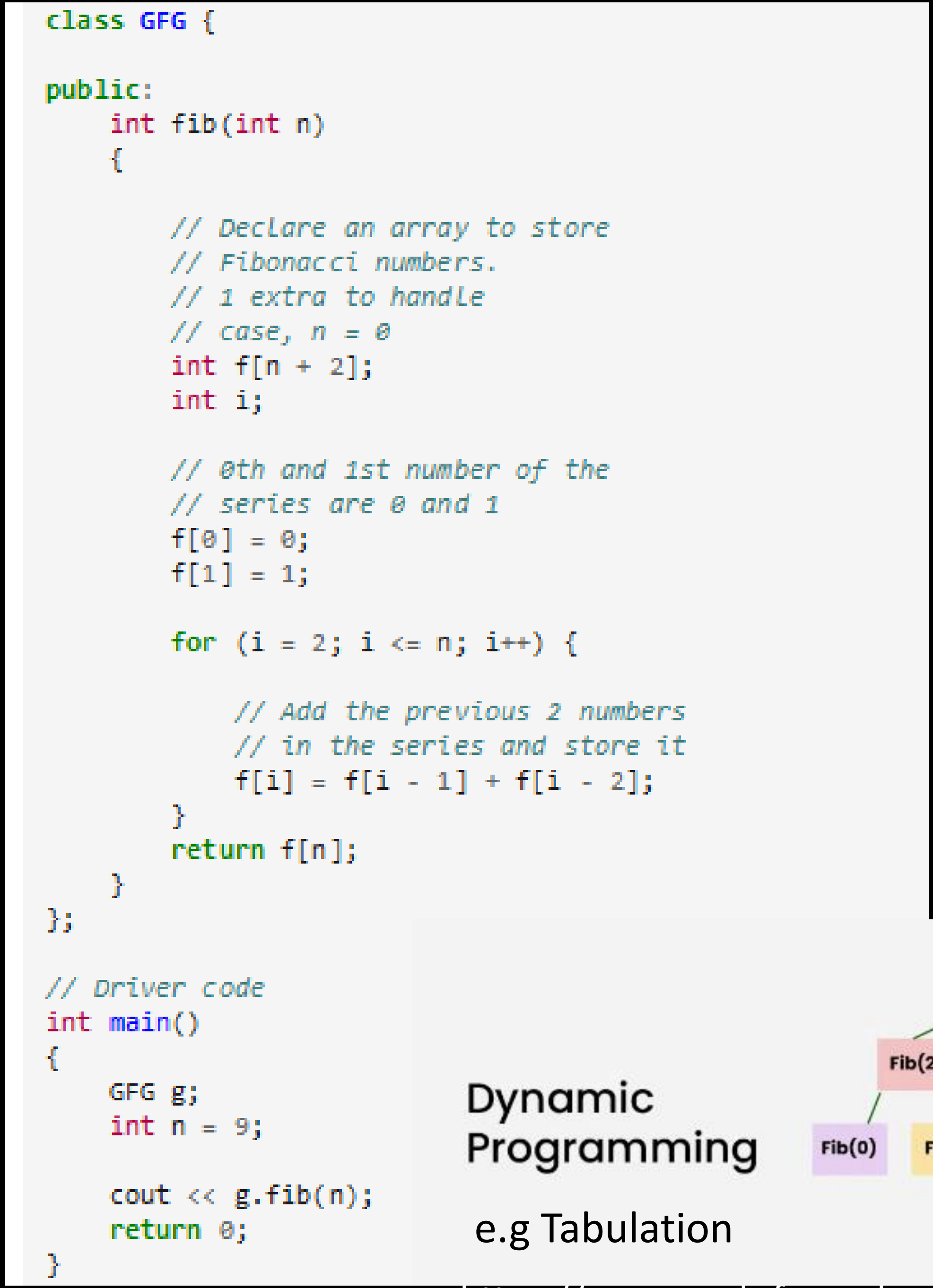
Here Same colurs denotes overlapping subproblems

Number of calls to Fib using recursion without Dynamic Programming

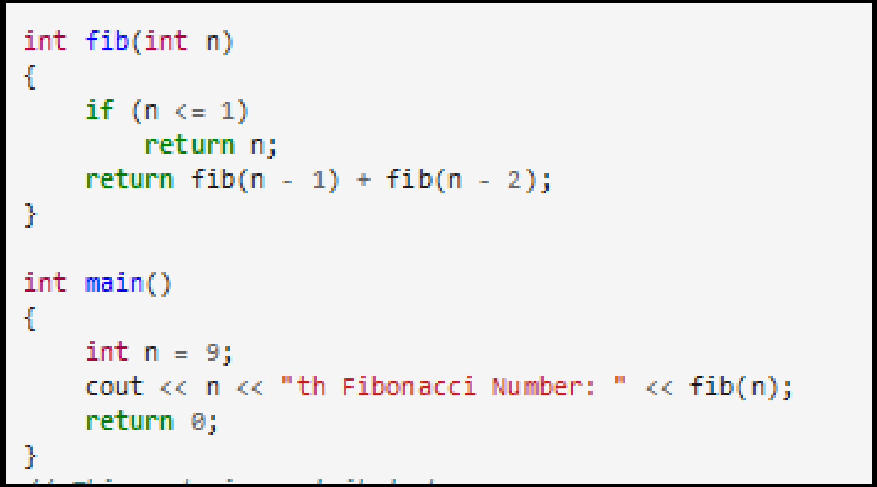


```
i                Fib(i)           numCalls
i = 0            1                1
i = 1            1                1
i = 2            2                3
i = 3            3                5
i = 4            5                9
i = 5            8                15
i = 6            13               25
i = 7            21               41
i = 8            34               67
i = 9            55               109
i = 10           89               177
i = 11           144              287
i = 12           233              465
i = 13           377              753
i = 14           610              1219
i = 15           987              1973
i = 16           1597             3193
i = 17           2584             5167
i = 18           4181             8361
i = 19           6765             13529
... snip ...
i = 50           20365011074      40730022147
```

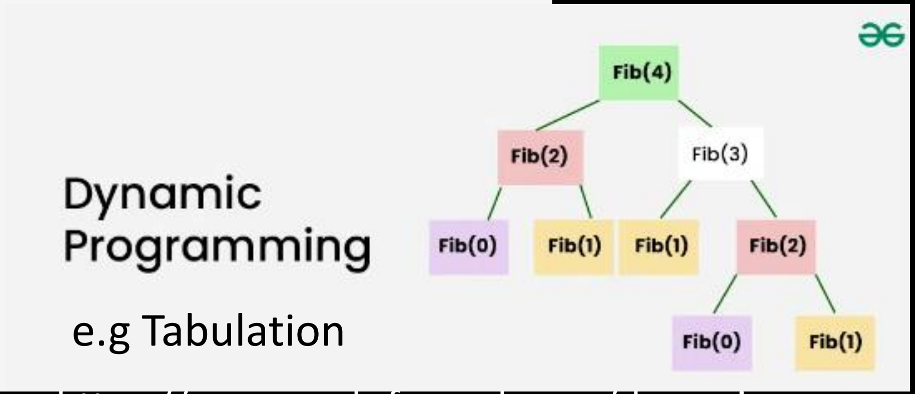
Fibonacci Numbers with and without Dynamic Programming – Simplified



```
class GFG {

public:
    int fib(int n)
    {

        // Declare an array to store
        // Fibonacci numbers.
        // 1 extra to handle
        // case, n = 0
        int f[n + 2];
        int i;

        // 0th and 1st number of the
        // series are 0 and 1
        f[0] = 0;
        f[1] = 1;

        for (i = 2; i <= n; i++) {

            // Add the previous 2 numbers
            // in the series and store it
            f[i] = f[i - 1] + f[i - 2];
        }
        return f[n];
    }
};

// Driver code
int main()
{
    GFG g;
    int n = 9;

    cout << g.fib(n);
    return 0;
}
```



```
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}

int main()
{
    int n = 9;
    cout << n << "th Fibonacci Number: " << fib(n);
    return 0;
}
```

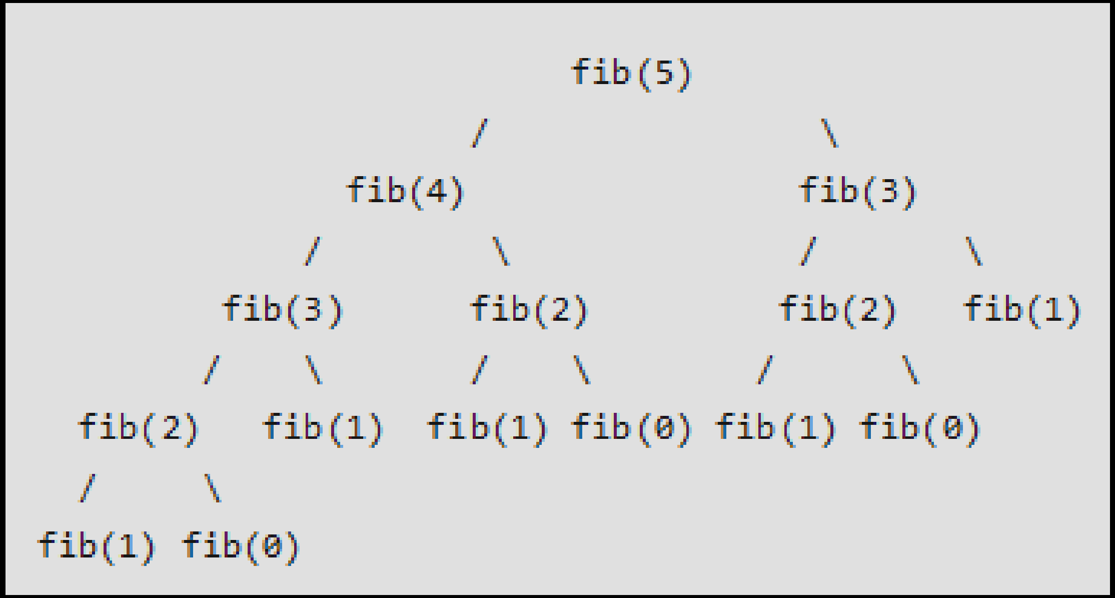


e.g Tabulation

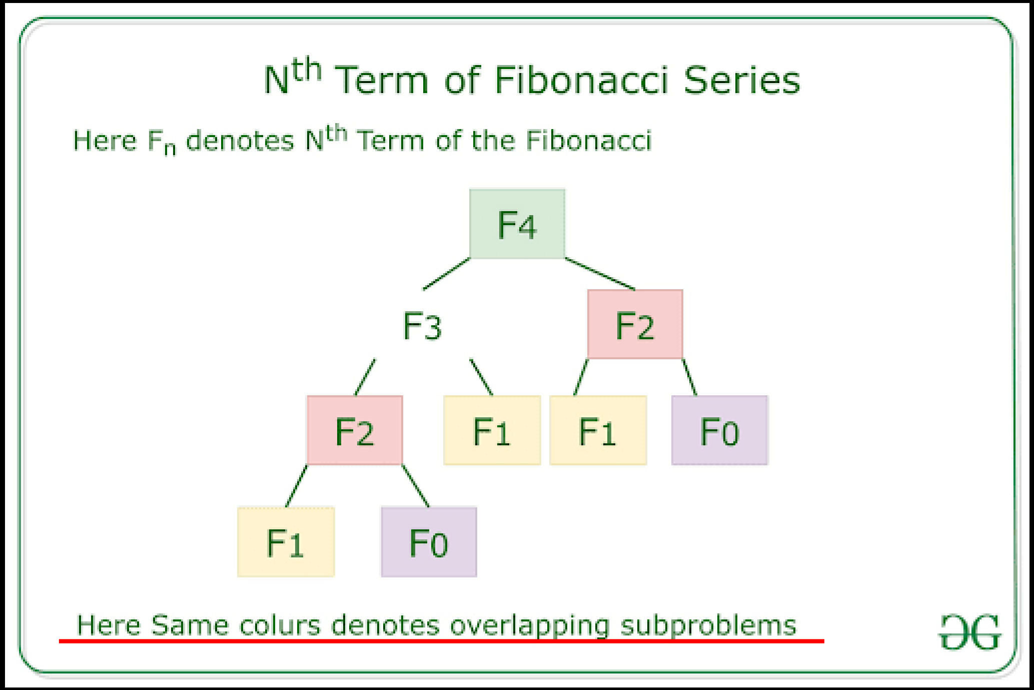
Recursive Approach (Naive): This directly translates the definition into a recursive function.

https://www.geeksforgeeks.org/dynamic-programming/

Fibonacci Numbers With and without Dynamic Programming



```
                         fib(5)
                     /             \
               fib(4)                fib(3)
             /        \              /     \
         fib(3)      fib(2)         fib(2)    fib(1)
        /    \        /    \       /      \
   fib(2)   fib(1)  fib(1) fib(0) fib(1) fib(0)
   /    \
fib(1) fib(0)
```


Nth Term of Fibonacci Series
Here F$_n$ denotes Nth Term of the Fibonacci
F4
F3
F2
F2
F1
F1
F0
F1
F0
Here Same colurs denotes overlapping subproblems

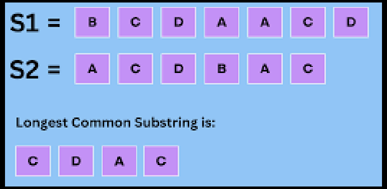
The same method call is being done multiple times for the same value. This can be optimized with the help of Dynamic Programming.

We can avoid the repeated work done in the Recursion approach by storing the Fibonacci numbers calculated so far.

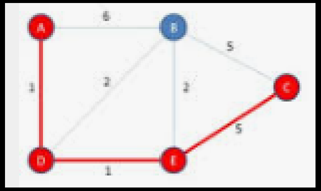
https://www.geeksforgeeks.org/dynamic-programming/

Example problems for DP

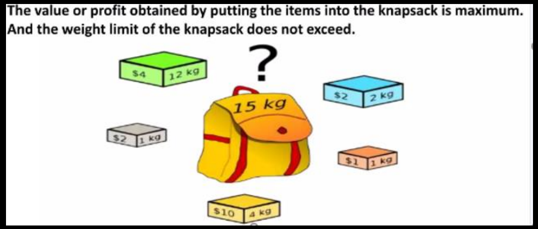
- Example 1: By using DP, we can efficiently calculate the Fibonacci sequence without having to recompute subproblems.
- Example 2: Longest common subsequence (finding the longest subsequence that is common to two strings)
    - Ref : https://www.youtube.com/watch?v=jHGgXV27qtk



- Example 3: Shortest path in a graph (finding the shortest path between two nodes in a graph)



- Example 4: Knapsack problem (finding the maximum value of items that can be placed in a knapsack with a given capacity)


The value or profit obtained by putting the items into the knapsack is maximum. And the weight limit of the knapsack does not exceed.

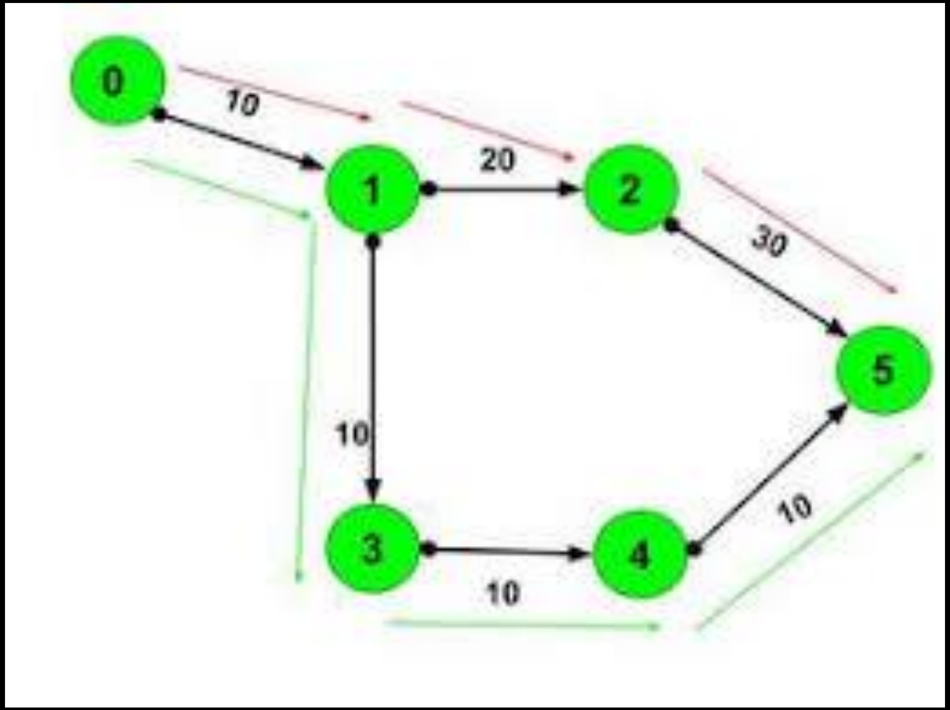
https://www.geeksforgeeks.org/dynamic-programming/

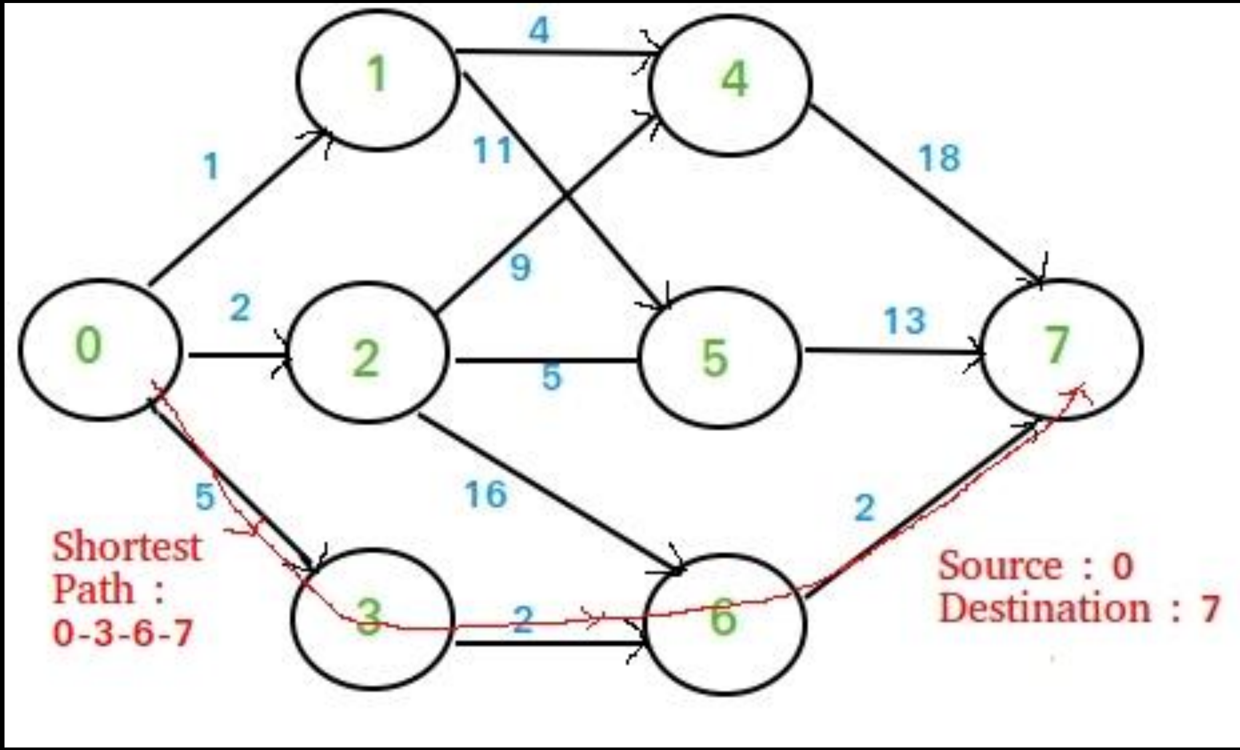
When to Use Dynamic Programming (DP)?

- Dynamic programming is an optimization technique used when solving problems that consists of the following characteristics:
- **1. Optimal Substructure:**
- Optimal substructure means that we combine the optimal results of subproblems to achieve the optimal result of the bigger problem.
- Example:
- Consider the problem of finding the minimum cost path in a weighted graph from a source node to a destination node. We can break this problem down into smaller subproblems:
    1. Find the minimum cost path from the source node to each intermediate node.
    2. Find the minimum cost path from each intermediate node to the destination node.
- The solution to the larger problem (finding the minimum cost path from the source node to the destination node) can be constructed from the solutions to these smaller subproblems.

https://www.geeksforgeeks.org/dynamic-programming/

What is Shortest Path?





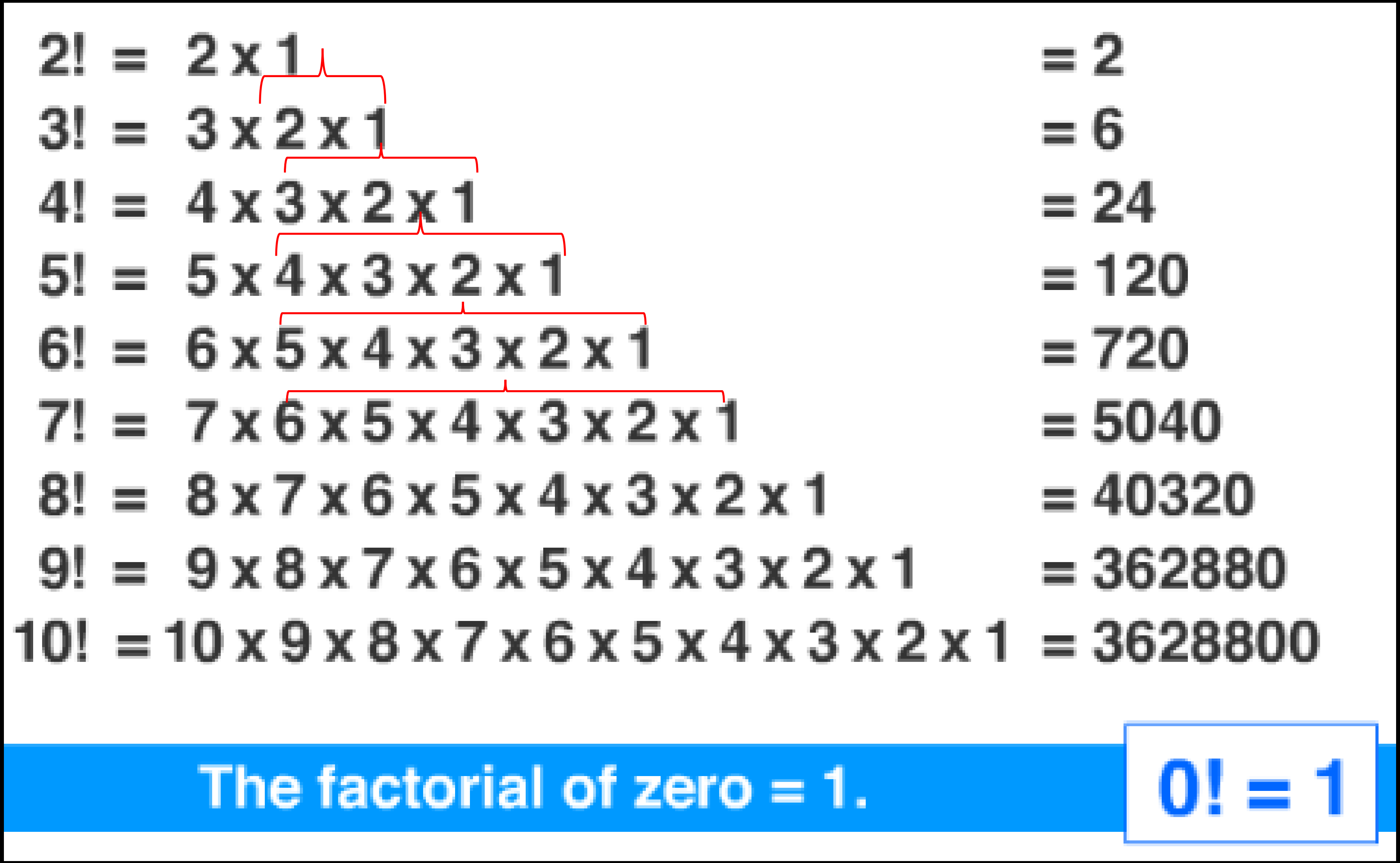
- For example, the Shortest Path problem has the following optimal substructure property –
  - If a node x lies in the shortest path from a source node u to destination node v, then the shortest path from u to v is the combination of the shortest path from u to x, and the shortest path from x to v.
- The standard All Pair Shortest Path algorithms like Floyd-Warshall and Bellman-Ford are typical examples of Dynamic Programming.

When to Use Dynamic Programming (DP)? Ctd…

- 2. Overlapping Subproblems:
- The same subproblems are solved repeatedly in different parts of the problem.
- Example:
  - Consider the problem of computing the Fibonacci series.
    - To compute the Fibonacci number at index n, we need to compute the Fibonacci numbers at indices n-1 and n-2.
    - This means that the subproblem of computing the Fibonacci number at index n-1 is used twice in the solution to the larger problem of computing the Fibonacci number at index n.

https://www.geeksforgeeks.org/dynamic-programming/

example Overlapping Subproblems: Factorial



$2! = 2 \times 1 = 2$

$3! = 3 \times \overbrace{2 \times 1} = 6$

$4! = 4 \times \overbrace{3 \times 2 \times 1} = 24$

$5! = 5 \times \overbrace{4 \times 3 \times 2 \times 1} = 120$

$6! = 6 \times \overbrace{5 \times 4 \times 3 \times 2 \times 1} = 720$

$7! = 7 \times \overbrace{6 \times 5 \times 4 \times 3 \times 2 \times 1} = 5040$

$8! = 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 40320$

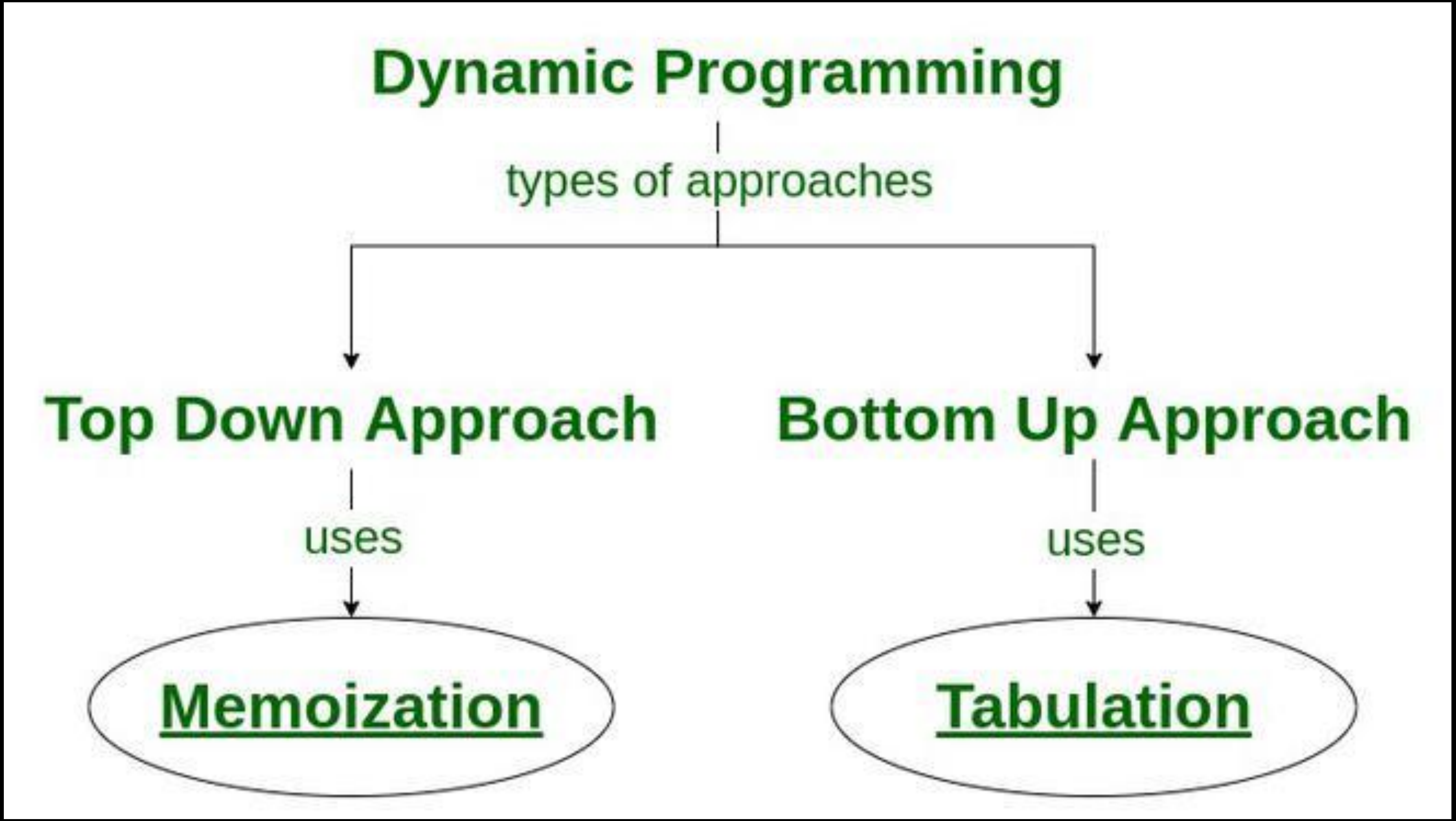
$9! = 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 362880$

$10! = 10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 3628800$

The factorial of zero = 1.

$0! = 1$

Approaches/Methods of Dynamic Programming (DP)



https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/

Approaches of Dynamic Programming (DP)

Dynamic programming can be achieved using two approaches:

- 1. Top-Down Approach (Memoization):
  - In the top-down approach, also known as memoization, we start with the final solution and recursively break it down into smaller subproblems. To avoid redundant calculations, we store the results of solved subproblems in a memoization table.
  - Let us breakdown Top down approach:
    1. Starts with the final solution and recursively breaks it down into smaller subproblems.
    2. Stores the solutions to subproblems in a table to avoid redundant calculations.
    3. Suitable when the number of subproblems is large and many of them are reused.

https://www.geeksforgeeks.org/dynamic-programming/

Approaches of Dynamic Programming (DP) ctd..

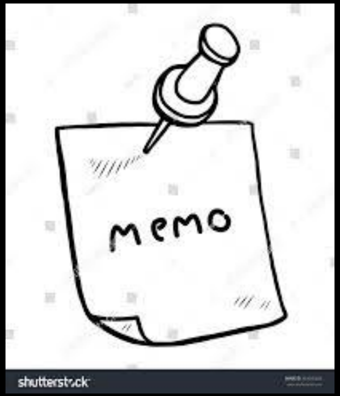
- 2. Bottom-Up Approach (Tabulation):
  - In the bottom-up approach, also known as tabulation, we start with the smallest subproblems and gradually build up to the final solution.
  - We store the results of solved subproblems in a table to avoid redundant calculations.
  - Let us breakdown Bottom-up approach:
    1. Starts with the smallest subproblems and gradually builds up to the final solution.
    2. Fills a table with solutions to subproblems in a bottom-up manner.
    3. Suitable when the number of subproblems is small and the optimal solution can be directly computed from the solutions to smaller subproblems.

https://www.geeksforgeeks.org/dynamic-programming/

	Tabulation ⬆	Memoization ⬇
State	State Transition relation is difficult to think	State transition relation is easy to think
Code	Code gets complicated when lot of conditions are required	Code is easy and less complicated
Speed	Fast, as we directly access previous states from the table	Slow due to lot of recursive calls and return statements
Subproblem solving	If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor	If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required
Table Entries	In Tabulated version, starting from the first entry, all entries are filled one by one	Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version. The table is filled on demand.

https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/

Memoization



- The term “Memoization” comes from the Latin word “memorandum” (to remember), which is commonly shortened to “memo” in American English, and which means “to transform the results of a function into something to remember.”.
  - storing the result so you can use it next time instead of calculating the same thing again and again
- In computing, memoization is used to speed up computer programs by eliminating the repetitive computation of results, and by avoiding repeated calls to functions that process the same input.

memo

To: [illegible]
From: [illegible]
CC: [illegible]
Date: [illegible]
Re: [illegible]

Overview [illegible]

Company

https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/

Why is Memoization used?

- Memoization is a specific form of caching that is used in dynamic programming.
- The purpose of caching is to improve the performance of our programs and keep data accessible that can be used later.
    - It stores the previously calculated result of the subproblem and uses the stored result for the same subproblem.
    - This removes the extra effort to calculate again and again for the same problem.
- We already know that if the same problem occurs again and again, then that problem is recursive in nature.

https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/

Memoization

- Memoization is a top-down approach where we cache the results of function calls and return the cached result if the function is called again with the same inputs.
- Memoization is typically implemented using recursion and is well-suited for problems that have a relatively small set of inputs

https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/

Memoization example?

- Let us take an example where the same subproblem repeats again and again.
- Below is a recursive method for finding the factorial of a number:

```
int factorial(unsigned int n)
{
  if (n == 0)
     return 1;
  return n * factorial(n – 1);
}
```

https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/

Memoization example?

- What happens if we use this recursive method?
- If you write the complete code for the above, you will notice that there will following in the code:

    factorial(n)

    main()
- Now if we have multiple queries to find the factorial, such as finding factorial of 2, 3, 9, and 5, then we will need to call the factorial() method 4 times:

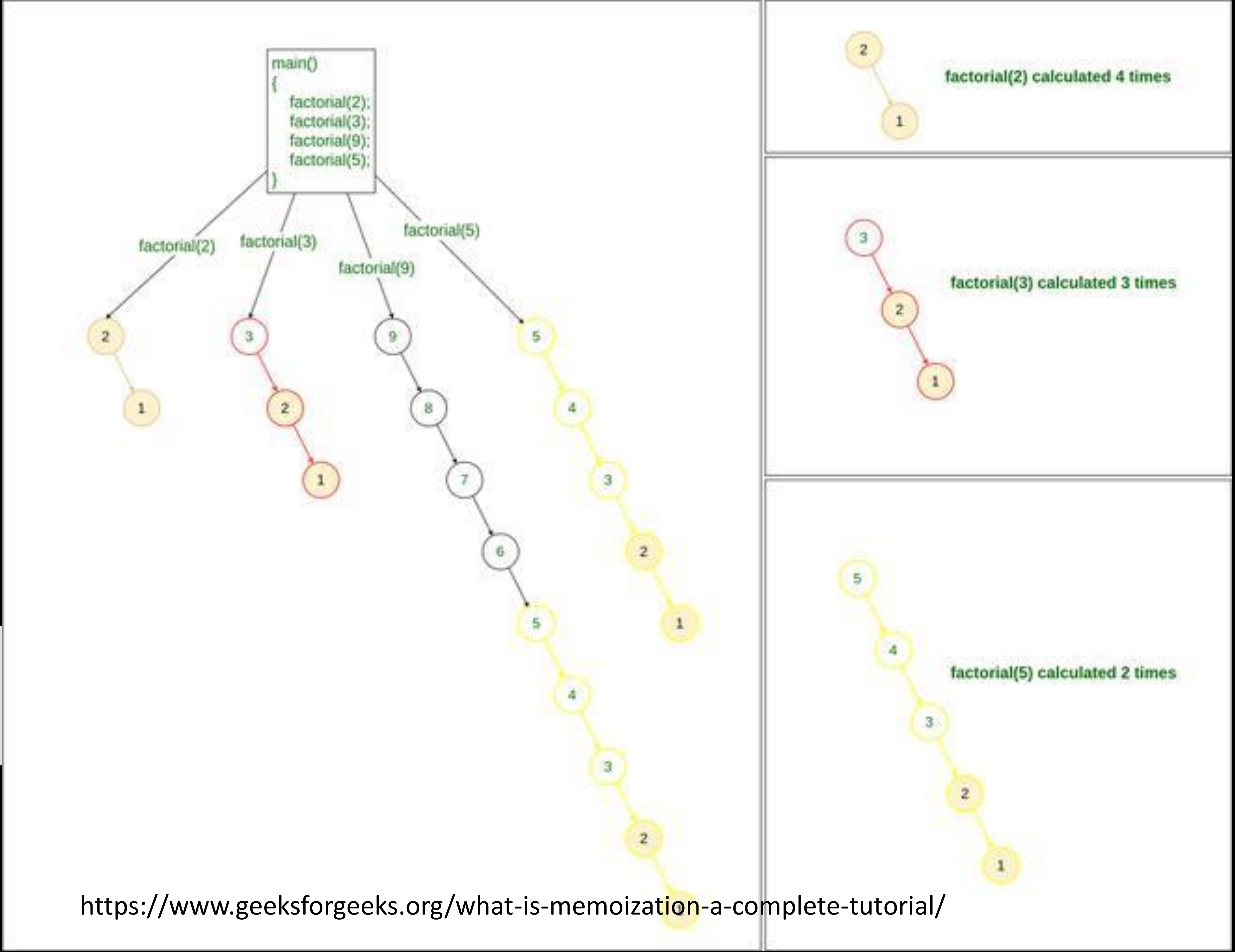
    factorial(2)

    factorial(3)

    factorial(9)

    factorial(5)

https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/



```
factorial(2)
factorial(3)
factorial(9)
factorial(5)
```

https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/

Memoization example?

- How Memoization can help optimization of problems?
  - If we notice in the above problem, while calculation factorial of 9:
    - We are calculating the factorial of 2
    - We are also calculating the factorial of 3,
    - and We are calculating the factorial of 5 as well
- Therefore if we store the result of each individual factorial at the first time of calculation, we can easily return the factorial of any required number in just O(1) time. This process is known as Memoization.

https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/

Memoization example?

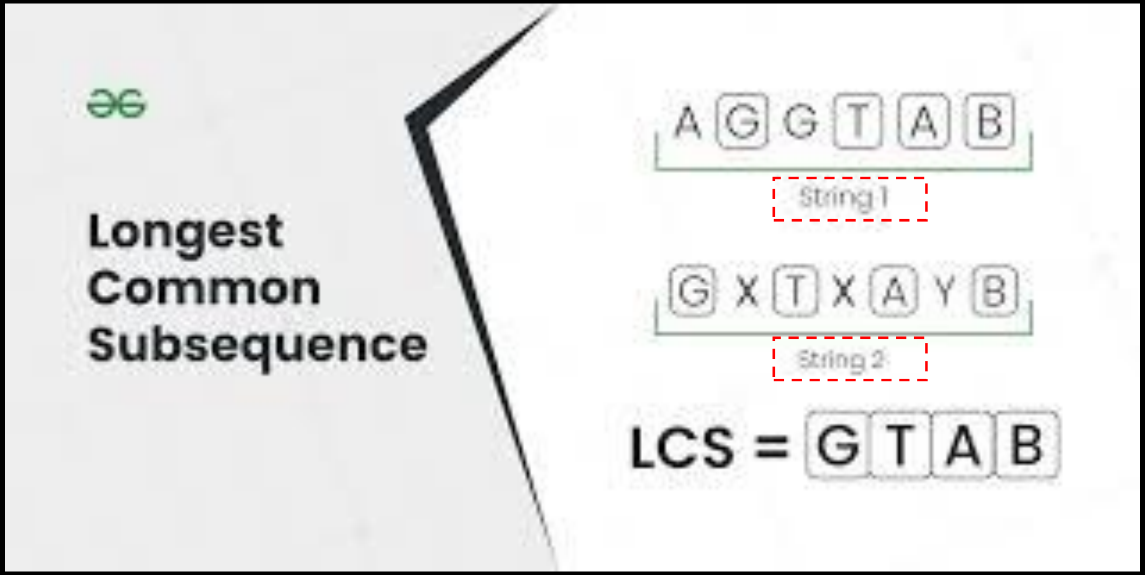
- Solution using Memoization
- How does memoization work?:
  - If we find the factorial of 9 first and store the results of individual sub-problems, we can easily print the factorial of each input in O(1).
  - Therefore the time complexity to find factorial numbers using memoization will be O(N)
    - O(N) to find the factorial of the largest input
    - O(1) to print the factorial of each input.

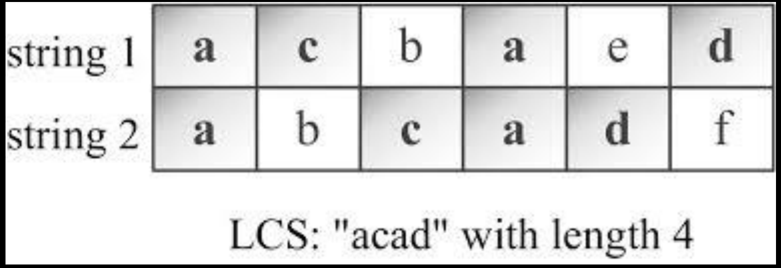
Types of Memoization: 1D,2D and 3D

- The Implementation of memoization depends upon the changing parameters that are responsible for solving the problem.
- There are various dimensions of caching that are used in memoization technique, Below are some of them:
  - 1D Memoization: The recursive function that has only one argument whose value was not constant after every function call.
    - e.g Fibonacci series
  - 2D Memoization: The recursive function that has only two arguments whose value was not constant after every function call.
    - e.g computing the Length of computing the Length of LCS for two strings
  - 3D Memoization: The recursive function that has only three arguments whose values were not constant after every function call.
    - e.g computing the Length of LCS for three strings

https://www.tutorialspoint.com/memorization-1d-2d-and-3d-dynamic-programming-in-java
https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/

What is LCS referred in previous slide?

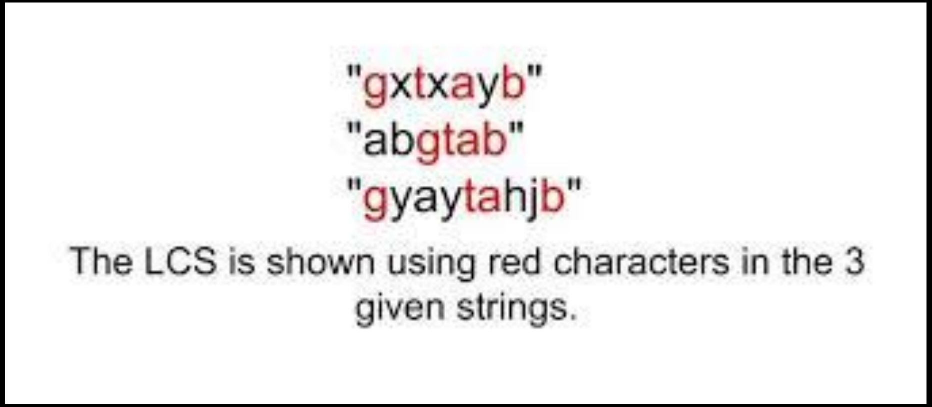




string 1	a	c	b	a	e	d
string 2	a	b	c	a	d	f

LCS: "acad" with length 4

Two Stings Example

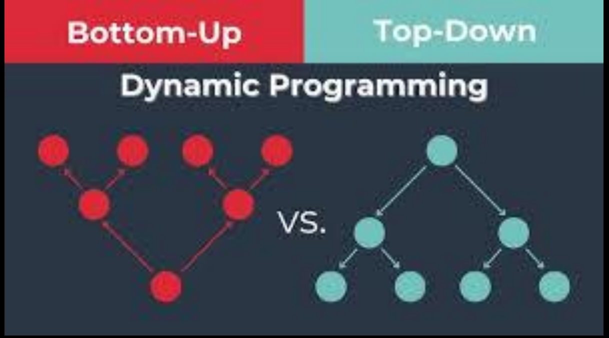


"gxtxayb"
"abgtab"
"gyaytahjb"

The LCS is shown using red characters in the 3 given strings.

Three Stings Example

Tabulation vs Memoization



- Memoization (top-down cache filling) refers to the technique of caching and reusing previously computed results.
- The memoized fib function would thus look like below:

```
memFib(n) {
    if (mem[n] is undefined)
        if (n < 2) result = n
        else result = memFib(n-2) + memFib(n-1)
        mem[n] = result
    return mem[n]
}
```

Usually does not fill all entries in DP structure

- Tabulation (bottom-up cache filling) is similar but focuses on filling the entries of the cache.
- Computing the values in the cache is easiest done iteratively. The tabulation version of fib would look like below:

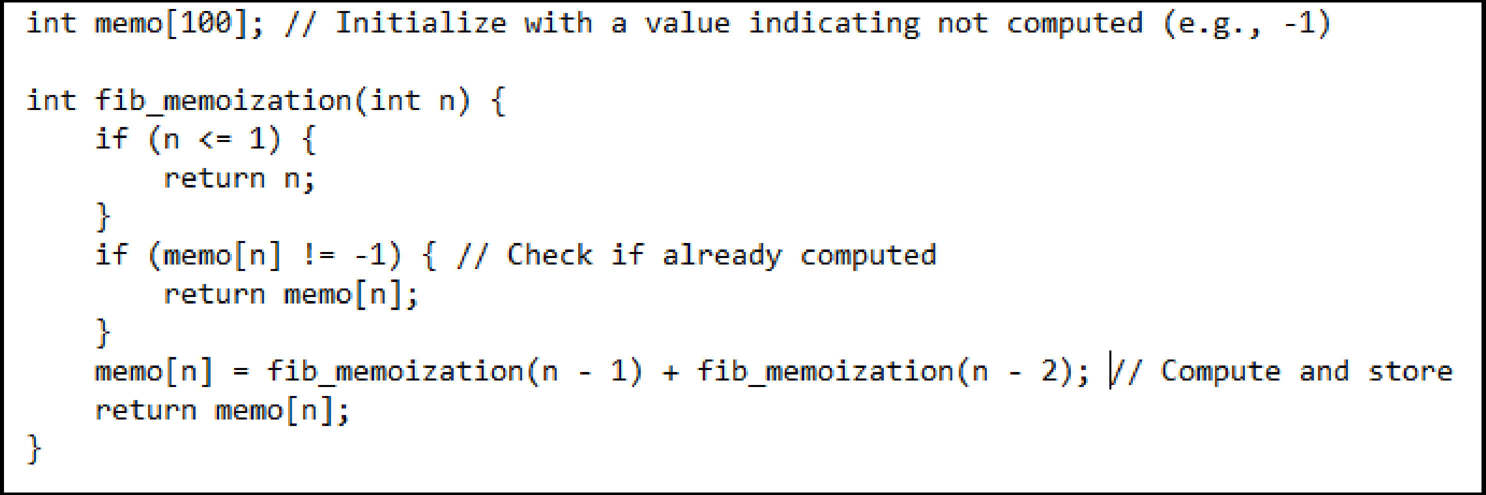
```
tabFib(n) {
    mem[0] = 0
    mem[1] = 1
    for i = 2...n
        mem[i] = mem[i-2] + mem[i-1]
    return mem[n]
}
```

Usually fills all entries in DP structure

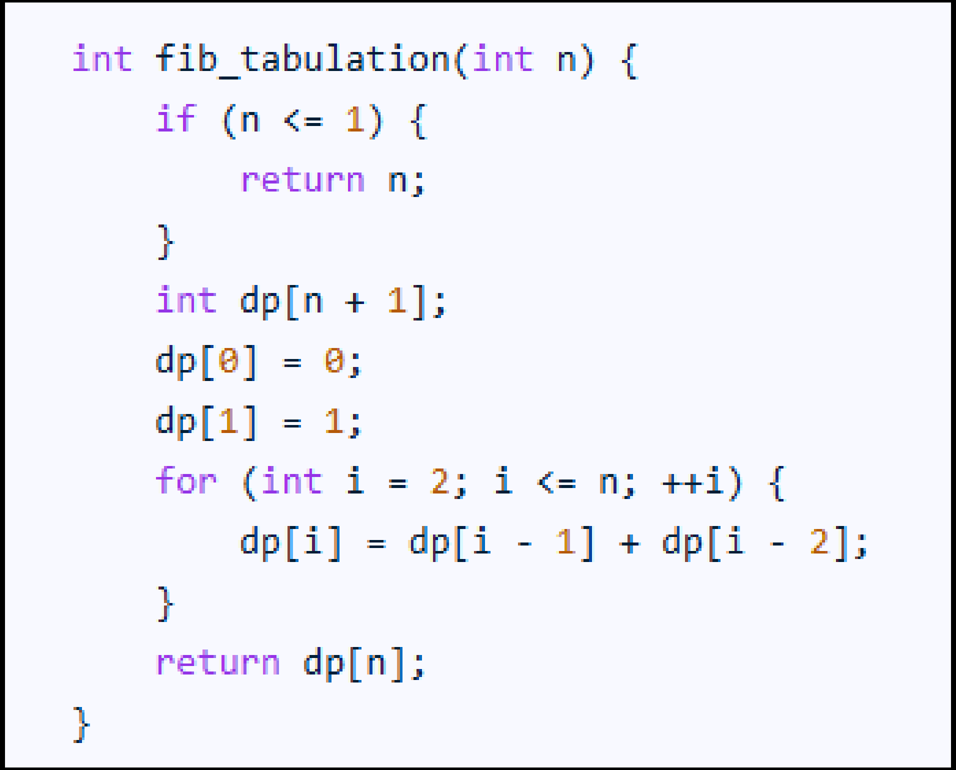
https://itnext.io/dynamic-programming-vs-divide-and-conquer-2fea680becbe

Memoization (Top-Down Dynamic Programming): This involves a recursive function with a data structure (e.g., an array or hash map) to store computed Fibonacci numbers

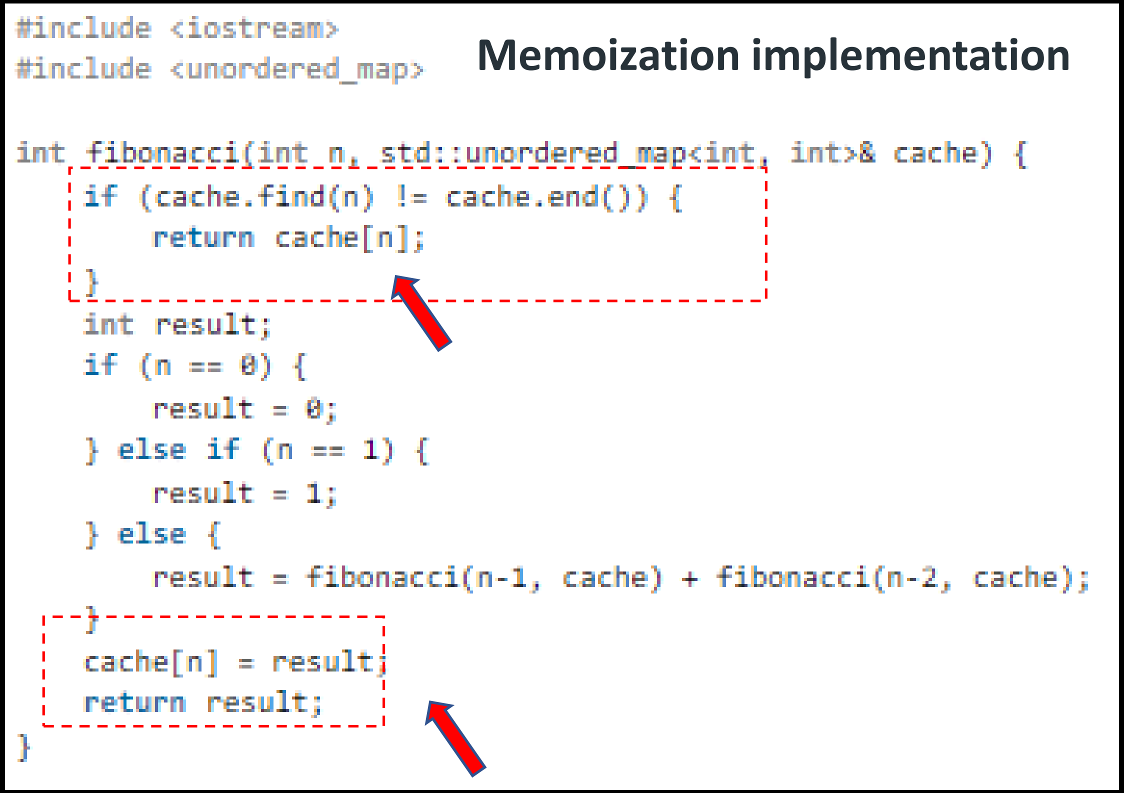
Tabulation (Bottom-Up Dynamic Programming): This builds up the solution iteratively by filling a table (array) from the base cases



```c
int memo[100]; // Initialize with a value indicating not computed (e.g., -1)

int fib_memoization(int n) {
    if (n <= 1) {
        return n;
    }
    if (memo[n] != -1) { // Check if already computed
        return memo[n];
    }
    memo[n] = fib_memoization(n - 1) + fib_memoization(n - 2); // Compute and store
    return memo[n];
}
```



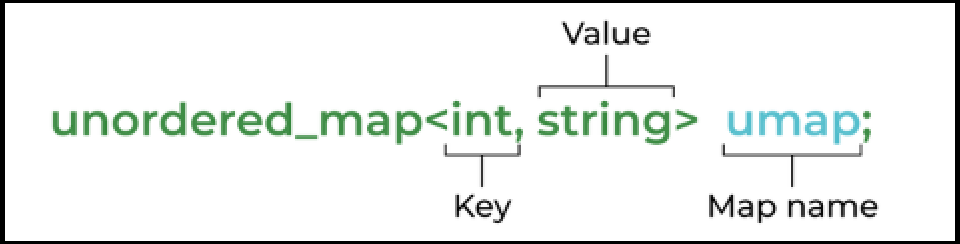
```c
int fib_tabulation(int n) {
    if (n <= 1) {
        return n;
    }
    int dp[n + 1];
    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; ++i) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    return dp[n];
}
```



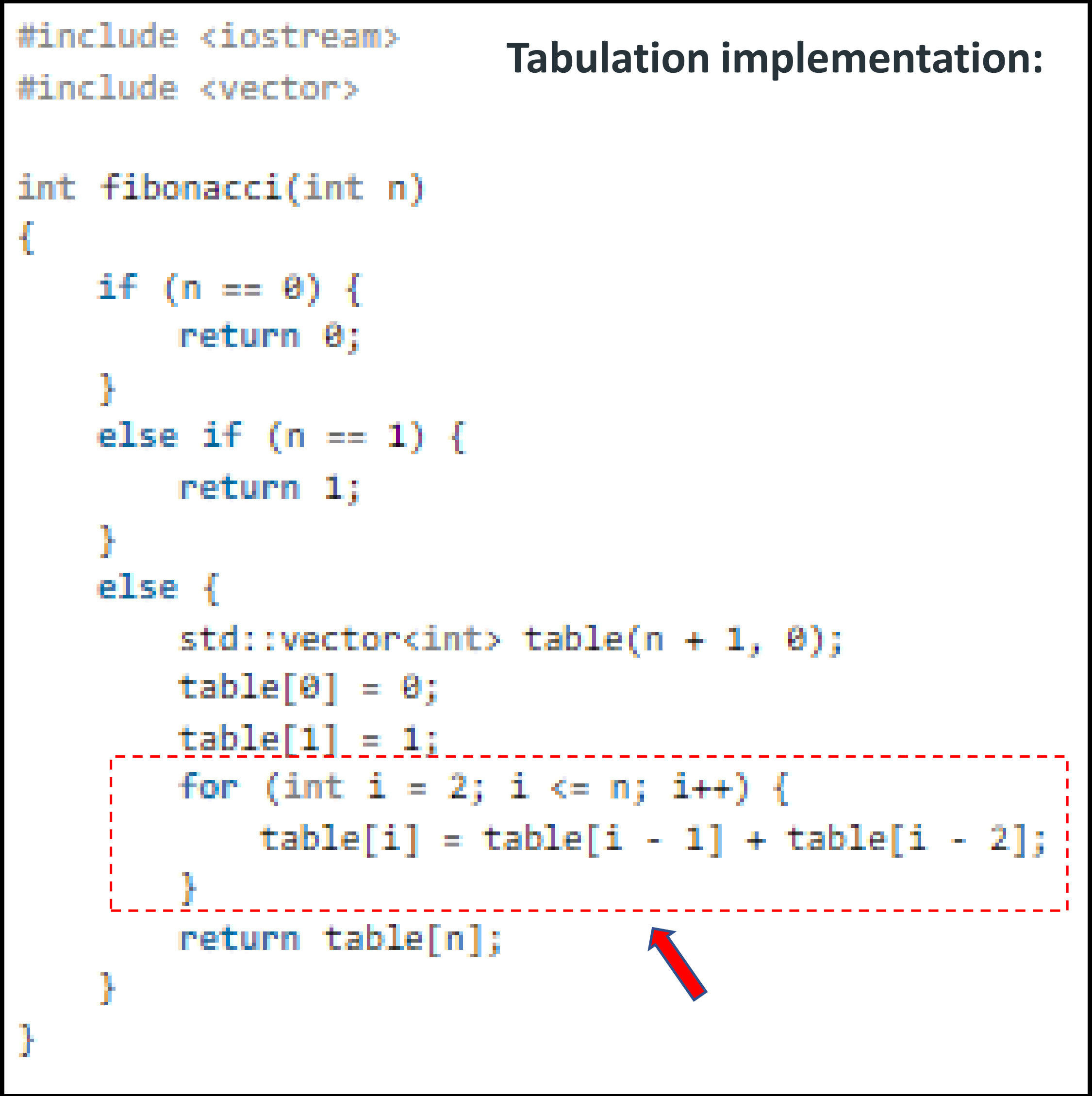
Memoization implementation

```cpp
#include <iostream>
#include <unordered_map>

int fibonacci(int n, std::unordered_map<int, int>& cache) {
    if (cache.find(n) != cache.end()) {
        return cache[n];
    }
    int result;
    if (n == 0) {
        result = 0;
    } else if (n == 1) {
        result = 1;
    } else {
        result = fibonacci(n-1, cache) + fibonacci(n-2, cache);
    }
    cache[n] = result;
    return result;
}
```

Memoization (top-down cache filling)



https://www.geeksforgeeks.org/unordered_map-in-cpp-stl/



Tabulation implementation:

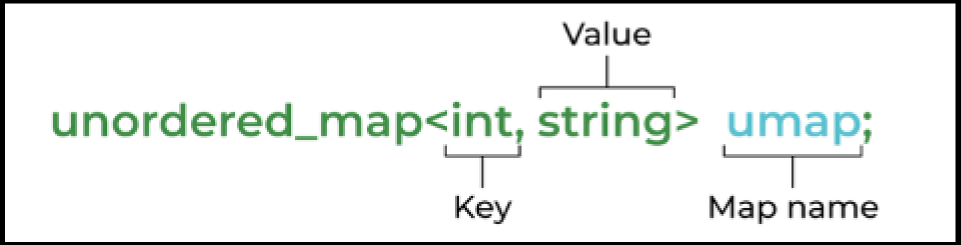
```cpp
#include <iostream>
#include <vector>

int fibonacci(int n)
{
    if (n == 0) {
        return 0;
    }
    else if (n == 1) {
        return 1;
    }
    else {
        std::vector<int> table(n + 1, 0);
        table[0] = 0;
        table[1] = 1;
        for (int i = 2; i <= n; i++) {
            table[i] = table[i - 1] + table[i - 2];
        }
        return table[n];
    }
}
```

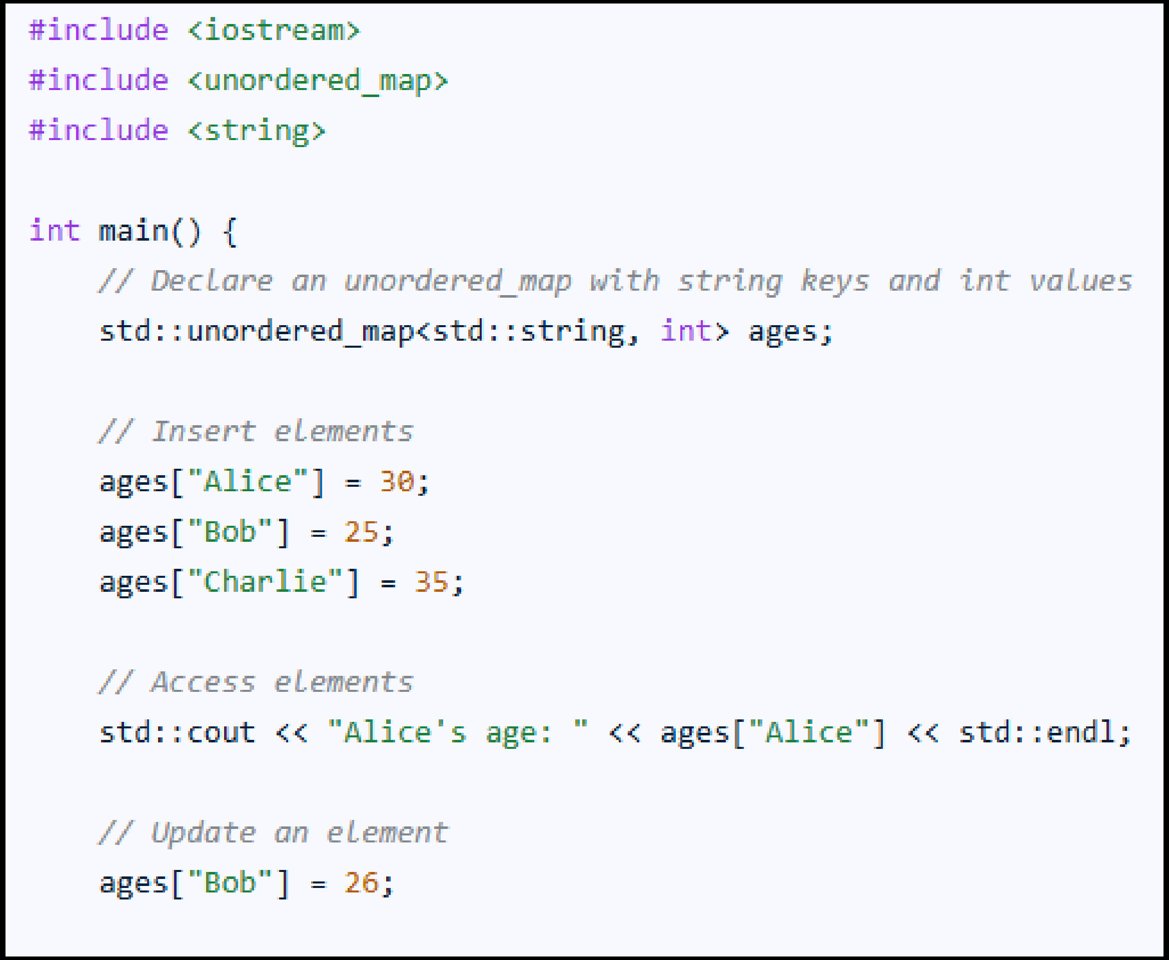
https://www.geeksforgeeks.org/tabulation-vs-memoization/

Tabulation (bottom-up cache filling)

unordered_map

- The unordered_map is a container in the C++ Standard Template Library (STL) that provides an associative array functionality.
- It stores elements in key-value pairs, where each key is unique and maps to a corresponding value.
- An associative array, also known as a map, dictionary, or hash table in various programming languages, is a data structure that stores data as a collection of key-value pairs. In the context of a phone book, an associative array is an ideal choice for organizing contact information due to its ability to efficiently store and retrieve data based on meaningful identifiers.





```cpp
#include <iostream>
#include <unordered_map>
#include <string>

int main() {
    // Declare an unordered_map with string keys and int values
    std::unordered_map<std::string, int> ages;

    // Insert elements
    ages["Alice"] = 30;
    ages["Bob"] = 25;
    ages["Charlie"] = 35;

    // Access elements
    std::cout << "Alice's age: " << ages["Alice"] << std::endl;

    // Update an element
    ages["Bob"] = 26;
```

Tabulation vs Memoization

- In above examples
  - The memoization implementation, we use a *dictionary object* called *cache* to store the results of function calls, and we use recursion to compute the results.
  - The tabulation implementation, we use an *array* called *table* to store the results of subproblems, and we use iteration to compute the results.
- Both implementations achieve the same result, but the approach used is different.
- Memoization is a top-down approach that uses recursion, while tabulation is a bottom-up approach that uses iteration.

https://www.geeksforgeeks.org/tabulation-vs-memoization/

Calculating Fibonacci sequence using DP

- Dynamic Programming can be applied to efficiently calculate the Fibonacci sequence:
  1. Identify the subproblem: The subproblem in this case is calculating the Fibonacci number at a specific index.
  2. Build the solution incrementally: Start by solving the subproblems from the smallest index upwards. Store the solutions in an array or table for future use.
  3. Utilize the stored solutions: When calculating a Fibonacci number at a particular index, retrieve the solutions of its preceding Fibonacci numbers from the table and use them to calculate the current Fibonacci number.

https://testbook.com/key-differences/difference-between-divide-and-conquer-and-dynamic-programming

Calculating Fibonacci sequence using DP

- By storing and reusing the solutions to subproblems, Dynamic Programming avoids redundant computations and greatly improves the efficiency of calculating Fibonacci numbers.
    - For example, to find the Fibonacci number at index 5, the Dynamic Programming approach would first calculate the Fibonacci numbers at indices 0, 1, 2, 3, and 4, storing them in a table.
    - Then, using these pre-computed values, it can directly calculate the Fibonacci number at index 5 without recalculating the entire sequence.
- Dynamic Programming significantly reduces the time complexity from exponential ($O(2^n)$) in a naive recursive approach to linear ($O(n)$) in the Fibonacci sequence calculation.

https://testbook.com/key-differences/difference-between-divide-and-conquer-and-dynamic-programming

Tabulation

- Tabulation is a bottom-up approach where we store the results of the subproblems in a table and use these results to solve larger subproblems until we solve the entire problem.
- Tabulation is typically implemented using iteration and is well-suited for problems that have a large set of inputs.

Some Advantages and Disadvantages of DP?

- Advantages of Dynamic Programming:
    1. Optimal solutions.
    2. Efficiency through avoiding redundant computations.
    3. Problem simplification through breaking down complex problems.

- Disadvantages of Dynamic Programming :
    1. Increased memory usage.
    2. Complexity in implementation.
    3. Dependency on problem structure.
    4. Computation overhead.

https://testbook.com/key-differences/difference-between-divide-and-conquer-and-dynamic-programming

Applications of Dynamic Programming (DP)

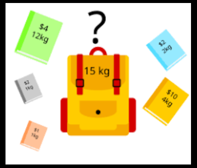
- Dynamic programming has a wide range of applications, including:
  - Optimization Domain: Knapsack problem, shortest path problem, maximum subarray problem
  - Computer Science Domain : Longest common subsequence, edit distance, string matching
  - Operations Research Domain : Inventory management, scheduling, resource allocation

https://www.geeksforgeeks.org/dynamic-programming/

Common Algorithms that Use Dynamic Programming:

i. Longest Common Subsequence (LCS): Finds the longest common subsequence between two strings.

ii. Shortest Path in a Graph: Finds the shortest path between two nodes in a graph.
   - All pair shortest path by Floyd-Warshall
   - Shortest path by Dijkstra

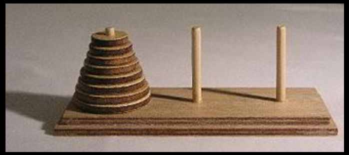


iii. Knapsack Problem: Determines the maximum value of items that can be placed in a knapsack with a given capacity.

iv. Matrix Chain Multiplication: Optimizes the order of matrix multiplication to minimize the number of operations.

v. Fibonacci Sequence: Calculates the nth Fibonacci number.

vi. Tower of Hanoi



vii. Project scheduling

https://www.geeksforgeeks.org/dynamic-programming/

What are the signs of DP suitability?

Identifying whether a problem is suitable for solving with dynamic programming (DP) involves recognizing certain signs or characteristics that suggest DP could be an effective approach. Here are some common signs that indicate a problem might be a good fit for dynamic programming:

1. Overlapping Subproblems: A problem that can be broken down into smaller subproblems that are solved independently, and the same subproblems encountered multiple times strongly indicates DP suitability.
2. Optimal Substructure: Problems that exhibit optimal substructure can often be solved using DP. This means that the optimal solution for a larger problem can be constructed from the optimal solutions of its smaller subproblems.
3. Recursive Nature: Problems that can be naturally expressed using recursion are often well-suited for DP.

https://www.masaischool.com/blog/understanding-dynamic-programming-101/

What are the signs of DP suitability? Ctd..

4. Memoization Opportunities: If you notice that you can improve a recursive algorithm by memoizing (caching) intermediate results, DP might be a good fit.
5. Sequential Dependencies: Problems where the solution depends on the results of previous steps or stages are often candidates for DP. DP is particularly useful when solving problems involving sequences, such as strings, arrays, or graphs.
6. Optimization or Counting: DP is often applied to optimization problems (maximizing or minimizing a value) or counting problems (finding the number of possible solutions).

https://www.masaischool.com/blog/understanding-dynamic-programming-101/

Some FAQ

- Is memoization the same as caching?
  - Memoization is actually a specific type of caching. The term caching can generally refer to any storing technique (like HTTP caching) for future use, but memoizing refers more specifically to caching function that returns the value.
- Why memoization is top-down?
  - The top-Down approach breaks the large problem into multiple subproblems. if the subproblem is solved already then reuse the answer. Otherwise, Solve the subproblem and store the result in some memory.
- Where is memoization used?
  - Memoization is an optimization technique used to speed up by caching the results of expensive function calls and returning them when the same inputs are encountered again.

https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/

FAQ

- Should I use tabulation or memoization?
  - For problems requiring all subproblems to be solved, tabulation typically outperforms memoization by a constant factor.
  - This is because the tabulation has no overhead of recursion which reduces the time for resolving the recursion call stack from the stack memory.
  - Whenever a subproblem needs to be solved for the original problem to be solved, memoization is preferable since a subproblem is solved lazily, i.e. only the computations that are required are carried out.

https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/

FAQ

- Why tabulation is faster than memoization?
    - Tabulation is usually faster than memoization, because it is iterative and solving subproblems requires no overhead of recursive calls.
    - Memoization is a technique used to speed up the execution of recursive or computationally expensive functions by caching the results of function calls and returning the cached results when the same inputs occur again.
    - The basic idea of memoization is to store the output of a function for a given set of inputs and return the cached result if the function is called again with the same inputs.
    - This technique can save computation time, especially for functions that are called frequently or have a high time complexity.

https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/

- Refer : **MIT. Dynamic Programming** https://www.youtube.com/watch?v=OQ5jsbhAv_M&list=PLfMspJ0TLR5HRFu2kLh3U4mvStMO8QURm&index=2