GUIDE DYNAMIC PROGRAMMING

ATALYK AKASH

The author of Dynamic Programming Patterns

Contents

What is Dynamic Programming

What is Dynamic Programming? Is it about programming dynamically? To understand the concept, we need to understand the meaning hidden behind the term Dynamic Programming.

Dynamic Programming was developed by Richard Bellman in the 1950s. The term refers to mathematical programming, which is also known as mathematical optimization. The name Dynamic Programming was proposed by Richard Bellman to use it as an umbrella for his works (Eye of the Hurricane: An Autobiography, 1984).

Meaning that Dynamic Programming is not about programming as it's known in the Computer Science world but about the mathematical optimization of a problem. Specifically, Dynamic Programming breaks a problem into sub-problems and optimizes sub-problems first, based on solutions for sub-problems, it builds the solution for the general problem. Thus, to eliminate the fear of words Dynamic Programming, we can think of it as substructural optimization. Dynamic Programming can be used in Computer Science as well as in Economics, Bioinformatics, etc.

Fibonacci Sequence

The Fibonacci Sequence is a classic example of the Dynamic Programming problem.

> The Fibonacci Sequence is the sequence where an element is a sum of two previous elements.

The following is a sample Fibonacci Sequence:

$0, 1, 1, 2, 3, 5, 8, 13, 21$ and so on.

Let's denote $F(n)$ as a function to find n^{th} Fibonacci number, then the general equation will be

$$F(n) = F(n-1) + F(n-2)$$

$$F(n) = \begin{cases} 0, & if\ n = 0 \\ 1, & if\ n = 1 \\ F(n-1) + F(n-2), & otherwise \end{cases}$$

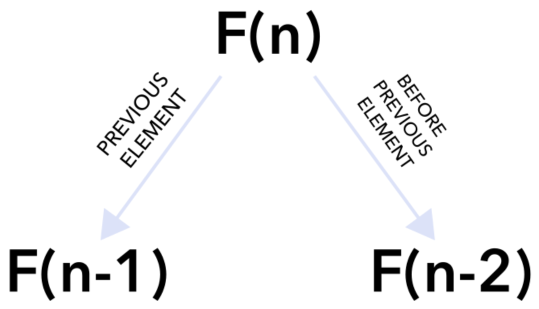

Figure 1. The recursive tree formula
for the Fibonacci Sequence.

To find 6^{th} Fibonacci number, find 5^{th} and 4^{th} Fibonacci numbers and sum them up.

$$F(6) = F(5) + F(4)$$

To find 5^{th} Fibonacci number, find 4^{th} and 3^{rd} Fibonacci numbers and sum them up.

$$F(5) = F(4) + F(3)$$

To find 4^{th} Fibonacci number, find 3^{rd} and 2^{nd} Fibonacci numbers and sum them up.

$$F(4) = F(3) + F(2)$$

To find 3^{rd} Fibonacci number, find 2^{nd} and 1^{st} Fibonacci numbers and sum them up.

$$F(3) = F(2) + F(1)$$

To find 2^{nd} Fibonacci number, find 1^{st} and 0 Fibonacci numbers and then sum them up.

$$F(2) = F(1) + F(0)$$

For 1^{st} and 0 Fibonacci numbers we already know the values.

$$F(1) = 1$$

$$F(0) = 0$$

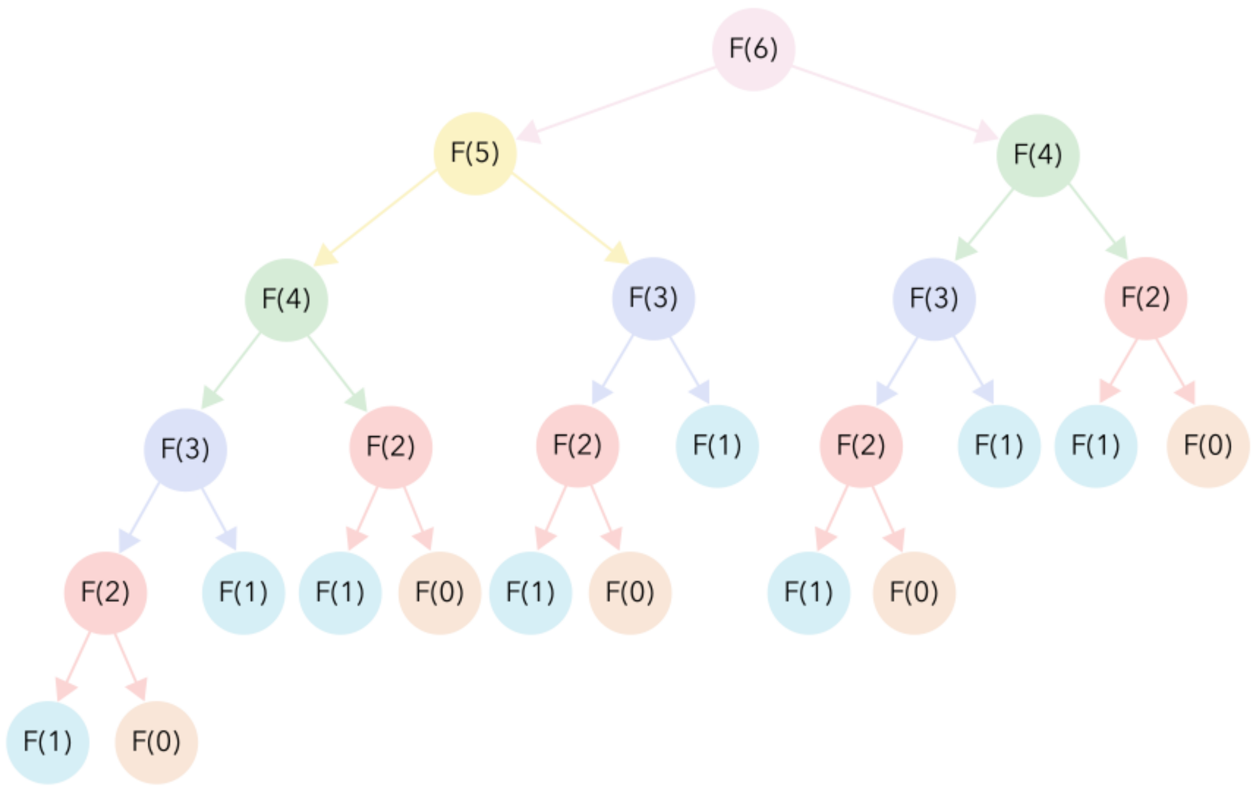

Figure 2. The recursive tree
for the Fibonacci Sequence.

Coin Change

Another example of the Dynamic Programming problem is the Coin Change Problem.

> Given a target amount and coins of different denominations, find the minimum number of coins to make the target amount.

Let's consider the following example

$$target\ amount\ =\ 7$$
$$coins\ =\ 1,3,5$$

From the above example, the minimum number of coins to make *7* is three. We can use two coins with denomination *1* and one coin with denomination *5,* so *7 = 5 + 1 + 1* (there is another option to choose *3 + 3 + 1*).

Let's denote $F(n)$ as a function to find the minimum number of coins to make n amount, then the general equation will be

$$F(n)\ =\ min\begin{pmatrix} F(n-5) \\ F(n-3) \\ F(n-1) \end{pmatrix} +\ 1$$

$$F(n) = \begin{cases} 0, & n = 0 \\ \min\big(F(n-5), F(n-3), F(n-1)\big) + 1, & otherwise \end{cases}$$

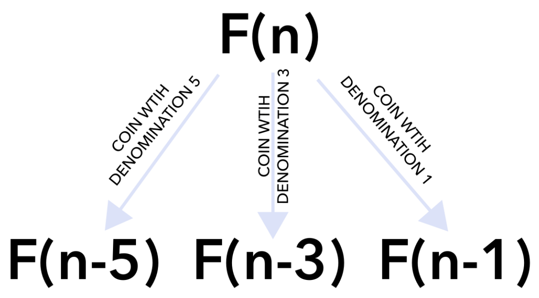

Figure 3. The recursive tree formula for the Coin Change Problem.

To find the minimum number of coins to make $F(7)$, we can choose one of the following options:

- use one coin with denomination 5 and remain with new $target\ amount\ 2$, $F(2)$
- use one coin with denomination 3 and remain with new $target\ amount\ 4$, $F(4)$
- use one coin with denomination 1 and remain with new $target\ amount\ 6$, $F(6)$

$$F(7) = min\begin{pmatrix} F(2)\ (use\ 5) \\ F(4)(use\ 3) \\ F(6)\ (use\ 1) \end{pmatrix} + 1$$

To make $F(2)$, choose one of the following options:

- can't use the coin with denomination 5 (greater than target amount)
- can't use the coin with denomination 3 (greater than target amount)
- use one coin with denomination 1 and remain with new $target\ amount\ 1$, $F(1)$

$$F(2) = min\begin{pmatrix} can't\ use\ 5\ or\ 3 \\ F(1)\ (use\ 1) \end{pmatrix} + 1$$

To make $F(1)$, choose one of the following options:

- can't use the coin with denomination 5 (greater than target amount)
- can't use the coin with denomination 3 (greater than target amount)
- use one coin with denomination 1 and remain with new $target\ amount\ 0$, $F(0)$

$$F(1) = min\begin{pmatrix} can't\ use\ 5\ or\ 3 \\ F(0)\ (use\ 1) \end{pmatrix} + 1$$

$$F(4) = min\begin{pmatrix} can't\ use\ 5 \\ F(1) \\ F(3) \end{pmatrix} + 1$$

$$F(1) = min\begin{pmatrix} can't\ use\ 5\ or\ 3 \\ F(0)\ (use\ 1) \end{pmatrix} + 1$$

$$F(3) = min\begin{pmatrix} can't\ use\ 5 \\ F(0) \\ F(2) \end{pmatrix} + 1$$

$$F(2) = min\begin{pmatrix} can't\ use\ 5\ or\ 3 \\ F(1) \end{pmatrix} + 1$$

$$F(1) = min\begin{pmatrix} can't\ use\ 5\ or\ 3 \\ F(0) \end{pmatrix} + 1$$

$$F(6) = min\begin{pmatrix} F(1) \\ F(3) \\ F(5) \end{pmatrix} + 1$$

$$F(1) = min\begin{pmatrix} can't\ use\ 5\ or\ 3 \\ F(0) \end{pmatrix} + 1$$

$$F(3) = min\begin{pmatrix} can't\ use\ 5 \\ F(0) \\ F(2) \end{pmatrix} + 1$$

$$F(2) = min\begin{pmatrix} can't\ use\ 5\ or\ 3 \\ F(1) \end{pmatrix} + 1$$

$$F(1) = min\begin{pmatrix} can't\ use\ 5\ or\ 3 \\ F(0) \end{pmatrix} + 1$$

$$F(5) = min\begin{pmatrix} F(0) \\ F(2) \\ F(4) \end{pmatrix} + 1$$

$$F(2) = min\begin{pmatrix} can't\ use\ 5\ or\ 3 \\ F(1) \end{pmatrix} + 1$$

$$F(1) = min\begin{pmatrix} can't\ use\ 5\ or\ 3 \\ F(0) \end{pmatrix} + 1$$

$$F(4) = min\begin{pmatrix} can't\ use\ 5 \\ F(1) \\ F(3) \end{pmatrix} + 1$$

$$F(1) = min\begin{pmatrix} can't\ use\ 5\ or\ 3 \\ F(0) \end{pmatrix} + 1$$

$$F(3) = min\begin{pmatrix} can't\ use\ 5 \\ F(0) \\ F(2) \end{pmatrix} + 1$$

$$F(2) = min\begin{pmatrix} can't\ use\ 5\ or\ 3 \\ F(1) \end{pmatrix} + 1$$

$$F(1) = min\begin{pmatrix} can't\ use\ 5\ or\ 3 \\ F(0) \end{pmatrix} + 1$$

$$F(0) = 0$$

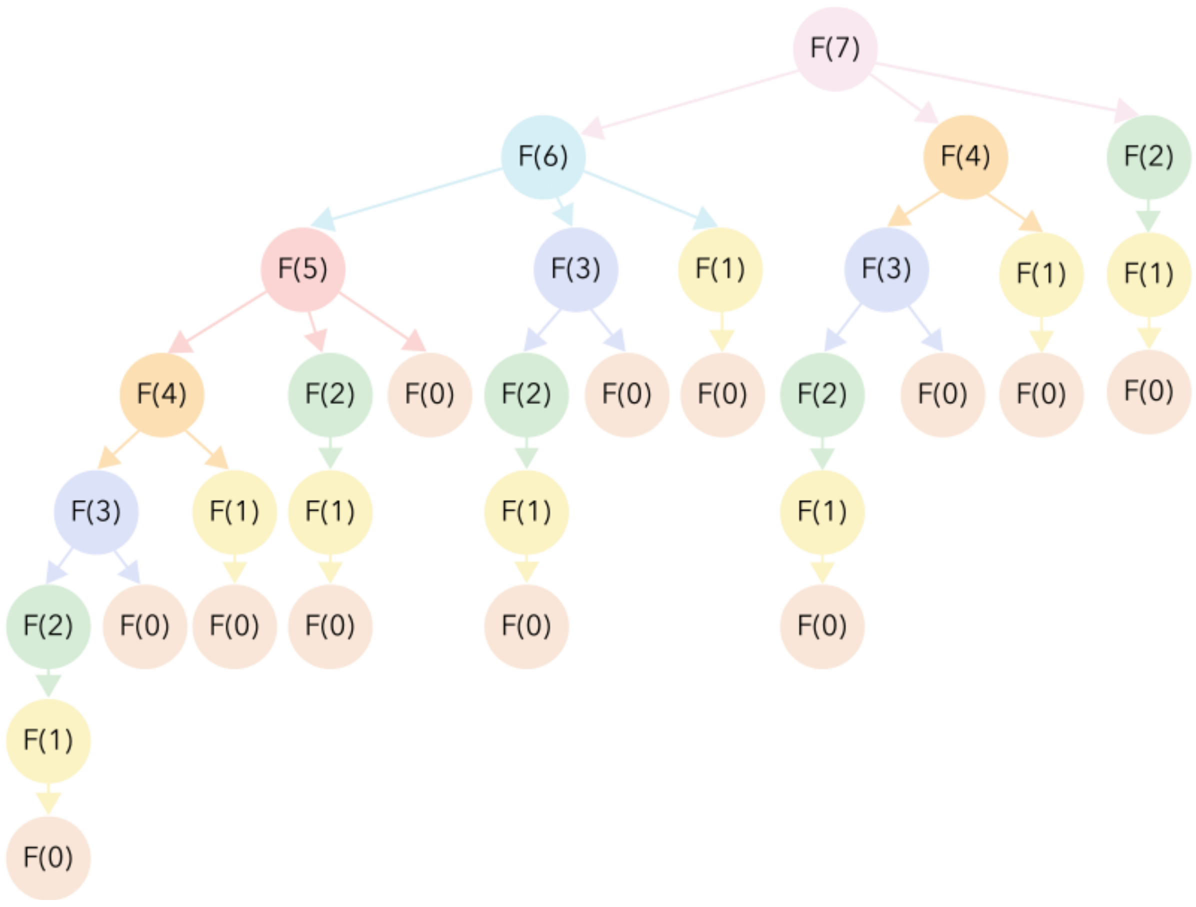

Figure 4. The recursive tree
for the Coin Change Problem.

Unique Paths

Given a 2D grid, find the total number of unique paths from the top-left $(0,0)$ corner to the bottom-right corner $(n-1, m-1)$, where n is the number of rows and m is the number of columns.

Let's denote $F(x,y)$ as a function to find the total number of unique paths to reach a position (x,y), then the general equation will be

$$F(x,y) = F(x-1,y) + F(x,y-1)$$

$$F(x,y) = \begin{cases} 1, x = 0 \ and \ y = 0 \\ F(x-1,y) + F(x,y-1), otherwise \end{cases}$$

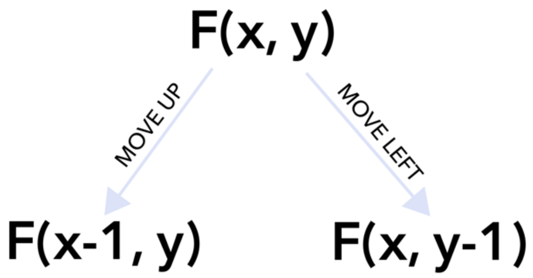

Figure 5. General recursive tree
for the Coin Change Problem.

For $n = 3$ and $m = 3$,

From position (2,2) we can go:

- up (1,2)
- left (2,1)

$$F(2,2) = F(1,2) + F(2,1)$$

From position (1,2) we can go:

- up (0,2)
- left (1,1)

$$F(1,2) = F(0,2) + F(1,1)$$

From position (0,2) we can go:

- only left (0,1)

$$F(0,2) = F(0,1)$$

From position (0, 1) we can go:

- only left (0,0)

$$F(0,1) = F(0,0)$$

$$F(2,1) = F(1,1) + F(2,0)$$

$$F(1,1) = F(0,1) + F(1,0)$$

$$F(0,1) = F(0,0)$$

$$F(1,0) = F(0,0)$$

$$F(2,0) = F(1,0)$$

$$F(1,0) = F(0,0)$$

$$F(0,0) = 1$$

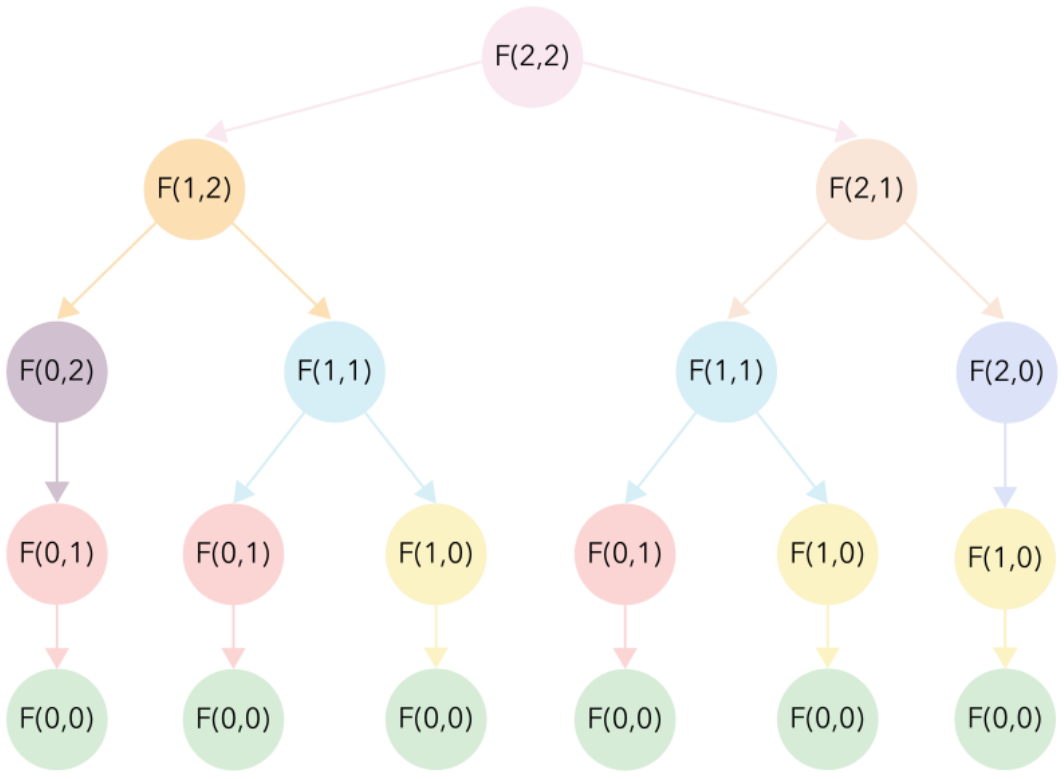

Figure 6. The recursive tree for Unique Paths.

From the recursive trees, we can see that the problem in the solutions above is the recalculation of the same sub-problems (states) multiple times, which leads to unnecessary extra work.

How does Dynamic Programming help us to solve this problem?

Dynamic Programming breaks a problem into sub-problems (states) and finds the most optimal solutions for all sub-problems. To avoid repetitive calculations, it caches already calculated solutions. Then based on solutions for sub-problems, it builds the solution for the general problem.

How to solve Dynamic Programming Problems

Dynamic Programming problems can be categorized into optimization and combinatorial problems. Optimization problems require finding the most optimal solution, such as the minimum/maximum/shortest/longest answer, while combinatorial problems require finding a number of ways/the probability of performing some operation.

In Computer Programming, there are two methods to solve Dynamic Programming problems:

- Top-Down (recursive)
- Bottom-Up (iterative)

Top-Down

The **top-down** approach solves a problem recursively, going from a target state to a base case and breaking the problem into sub-problems (states). Then it finds an optimal solution for every sub-problem and stores a result for reuse in the future.

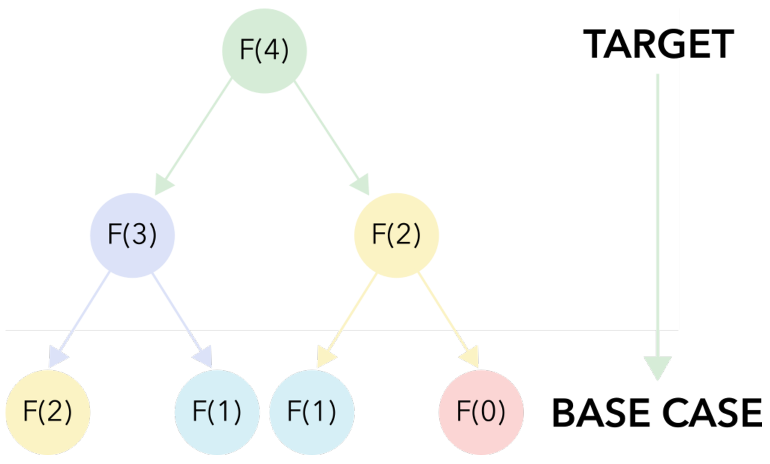

Figure 7. The top-down approach.

How to come up with a recursive solution

Perform the following steps to understand how to come up with a recursive solution.

Step 1. Iterate Over Options
When we solve a problem recursively, for every sub-problem (state), we have different options (directions) to choose to reach our target. We need to iterate over all possible options to find the most optimal solution.

$$F(n) \rightarrow \begin{matrix} F(option\ 1) \\ F(option\ 1) \\ \dots \\ F(option\ k) \end{matrix}$$

Fibonacci Sequence

$$F(current\ term) \rightarrow \begin{matrix} F(last\ term) \\ F(one\ before\ the\ last\ term) \end{matrix}$$

There are two options to choose to obtain n^{th} term:

- the last term
- one before the last term

Coin Change Problem

$$F(target) \rightarrow \begin{matrix} F(use\ coin\ with\ denomination\ 5) \\ F(use\ coin\ with\ denomination\ 3) \\ F(use\ coin\ with\ denomination\ 1) \end{matrix}$$

There are three options to choose to make the target amount:

- use the coin with denomination 5
- use the coin with denomination 3
- use the coin with denomination 1

Unique Paths

$$F(x,y) \rightarrow \begin{matrix} F(move\ up) \\ F(move\ left) \end{matrix}$$

There are two options (directions) to choose to reach the target position:

- move up
- move left

Step 2. Reduce Target
Choosing every option, we reduce a target to a particular value, and we call a recursive function again with the new updated target.

Fibonacci Sequence

$$F(n) \rightarrow \begin{matrix} F(n-1) \\ F(n-2) \end{matrix}$$

- Choosing the last term $F(n-1)$ reduces the problem to find $(n-1)^{th}$ term.
- Choosing the one before the last term $F(n-2)$ reduces the problem to find $(n-2)^{th}$ term.

$$F(2) \rightarrow \begin{matrix} F(1) \\ F(0) \end{matrix}$$

Coin Change Problem

$$F(target) \rightarrow \begin{matrix} F(target - 5) \\ F(target - 3) \\ F(target - 1) \end{matrix}$$

- Choosing the coin with denomination 5 reduces the problem to make the new target amount of $target - 5$.
- Choosing the coin with denomination 3 reduces the problem to make the new target amount of $target - 3$.
- Choosing the coin with denomination 1 reduces the problem to make the new target amount of $target - 1$.

$$F(11) \rightarrow \begin{matrix} F(11 - 5) \\ F(11 - 3) \\ F(11 - 1) \end{matrix}$$

$$F(11) \rightarrow \begin{matrix} F(6) \\ F(8) \\ F(10) \end{matrix}$$

Unique Paths

$$F(x,y) \rightarrow \begin{matrix} F(x-1,y) \\ F(x,y-1) \end{matrix}$$

- Choosing the option to move up reduces the problem to find the number of unique paths to reach $(x-1,y)$.
- Choosing the option to move left reduces the problem to find the number of unique paths to reach $(x,y-1)$.

$$F(2,2) \rightarrow \begin{matrix} F(1,2) \\ F(2,1) \end{matrix}$$

Step 3. Choose Optimal Solution
After coming from recursive calls, we choose an optimal solution among these calls and add the value for the current state.

Fibonacci Sequence

Sum up two previous values.

$$F(n) = F(n-1) + F(n-2)$$

Coin Change Problem

Choose the option that leads to the most optimal solution and add one to use one coin for the current state.

$$F(target) = min\begin{pmatrix} F(target - 5) \\ F(target - 3) \\ F(target - 1) \end{pmatrix} + 1$$

Unique Paths

Sum up both options to find the total number of distinct ways.

$$F(x,y) = F(x-1,y) + F(x,y-1)$$

Step 4. Base Case
When we reach a base case, we need to quit a recursive function. The base case can be a base state or case when a sub-problem is no longer valid (in this case, when it returns the worst result).

Fibonacci Sequence

We already know that the first two Fibonacci numbers are 0 and 1 respectively.

$$F(1) = 1$$

$$F(0) = 0$$

```
int F(n) {
    if (n == 0) {
        // base case
    }
    if (n == 1) {
        // base case
    }
}
```

Coin Change

When $target$ equals 0 (we don't need any coin to make 0) or less than 0, the function reaches the base case.

$$F(0) = 0$$

```
int F(target) {
    if (target == 0) {
        // base case
    }
    if (target < 0) {
        // not valid
    }
}
```

Unique Paths

When the position equals the destination cell or the position is out of bounds, the function reaches the base case.

$$F(0,0) = 1$$

```
int F(x, y) {
    if (x == 0 && y == 0) {
        // base case
    }
    if (x < 0 || y < 0) {
        // not valid
    }
}
```

Step 5. Return Result
Return a result of a recursive function.

After reaching the end of a function, return a result.

`return result`

Applying the steps above to solve examples recursively

Recursive solution for the Fibonacci Sequence

```
int Fib(int n) {
    if (n == 0) {
        return 0; // step 4
    }
    if (n == 1) {
        return 1; // step 4
    }

    int last = Fib(n-1); // step 1 & 2
    int beforeLast = Fib(n-2); // step 1 & 2
    int result = last + beforeLast; // step 3

    return result; // step 5
 }
```

Recursive solution for Coin Change Problem

```
int CoinChange(int target, vector<int> &coins) {
    if (target == 0) {
        return 0; // step 4
    }
    int result = inf;
    for (int i = 0; i < coins.size(); ++i) { // step 1
```

```
        if (coins[i] <= target) {
            int substateSolution = CoinChange(target-coins[i], coins); // step 2
            result = min(result, substateSolution+1); // step 3
        }
    }
    return result; // step 5
}
```

Recursive solution for Unique Paths

```
int UniquePaths(int x, int y) {
    if (x == 0 && y == 0) {
        return 1; // step 4
    }
    if (x < 0 || y < 0) {
        return 0; // step 4
    }
    int up = UniquePaths(x-1, y); // step 1 & 2
    int left = UniquePaths(x, y-1); // step 1 & 2

    int result = up + left; // step 3

    return result; // step 5
}
```

The problem with these recursive solutions is the recalculation of sub-problems, which leads to slow performance.

How to calculate the time complexity of a recursive solution

To calculate the time-complexity of a recursive solution, try to answer the following questions:

- How many times does a function call itself (t)?
- How many times is a function being recursed (k)?

Based on that, we can say that the time complexity of a plain recursive solution is exponential $O(t^k)$.

If we draw a recursive tree, we can find the time complexity by summing up the number of nodes in the tree.

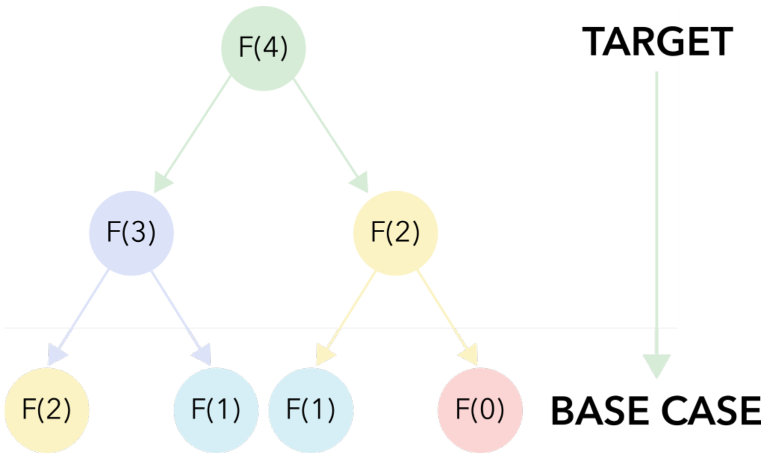

Figure 8. The recursive tree.

Fibonacci Sequence

To find n^{th} Fibonacci number, we need to sum up two previous Fibonacci numbers.

$$F(n) = F(n-1) + F(n-2)$$

The function is being called two times:

- $F(n-1)$
- $F(n-2)$

Then it's being recursed n times from n to 0, where n is the depth of the recursive tree for the function.

So, the time complexity of the plain recursive solution is $O(2^n)$.

Let's calculate the time complexity of the function to see if the statement above is true.

Let's denote $T(n)$ as a function to calculate the time complexity of the Fibonacci Sequence.

$$T(n) = T(n-1) + T(n-2) + C,$$

where $T(n-1)$ and $T(n-2)$ are the time complexities to find $n-1^{th}$ Fibonacci and $n-2^{th}$ Fibonacci numbers respectively, and C is the number of constant operations performed inside the recursive call.

For simplicity let's assume that $T(n-2) \leq T(n-1)$.

$$T(n) = 2 \times T(n-1) + C$$

$$T(n-1) = 2 \times T(n-2) + C$$

$$T(n) = 2 \times (2 \times T(n-2) + C) + C$$

$$T(n) = 2^2 \times T(n-2) + 3C$$

$$T(n-2) = 2 \times T(n-3) + C$$

$$T(n) = 2^2 \times (2 \times T(n-3) + C) + 3C$$

$$T(n) = 2^3 \times T(n-3) + 7C$$

$$T(n) = 2^k \times T(n-k) + (2^k - 1) \times C$$

$$T(0) = 1$$

$$n - k = 0$$

$$n = k$$

$$T(n) = 2^n \times T(0) + (2^n - 1) \times C = 2^n + (2^n - 1) \times C$$

The upper bound time complexity is $O(2^n)$.

Using the idea that $F(n)$ and $T(n)$ are calculated in the same way, we can find a tighter upper bound time complexity. Approximate value for $F(n)$ is $O(\phi^n)$, where $\phi = (1 + \sqrt{5})/2 \approx 1.618$.

Consequently,

$$T(n) = O(\phi^n) = O((1 + \sqrt{5})/2)^n) = O(1.618^n)$$

Coin Change

The function is being called n times:

- $F(target\ -\ coin\ 1)$
- $F(target\ -\ coin\ 2)$
- $F(target\ -\ coin\ 3)$
- $\dots$
- $F(target\ -\ coin\ n)$

where n is the number of coins.

Then it's being recursed $target\ amount$ times (worst case).

So, the time complexity of the plain recursive solution is $O(n^{target\ amount})$.

Unique Paths

The function is being called two times:

- $F(x-1,y)$
- $F(x,y-1)$

Then it's being recursed $n+m$ times, where n is the number of rows and m is the number of columns.

Why is the function being recursed $n+m$ times?

For every recursive call, we move one step up, or one step left. In other words, to reach leaf nodes, we need to exhaust total rows and total columns, meaning that the depth of our recursive tree will be $n+m$.

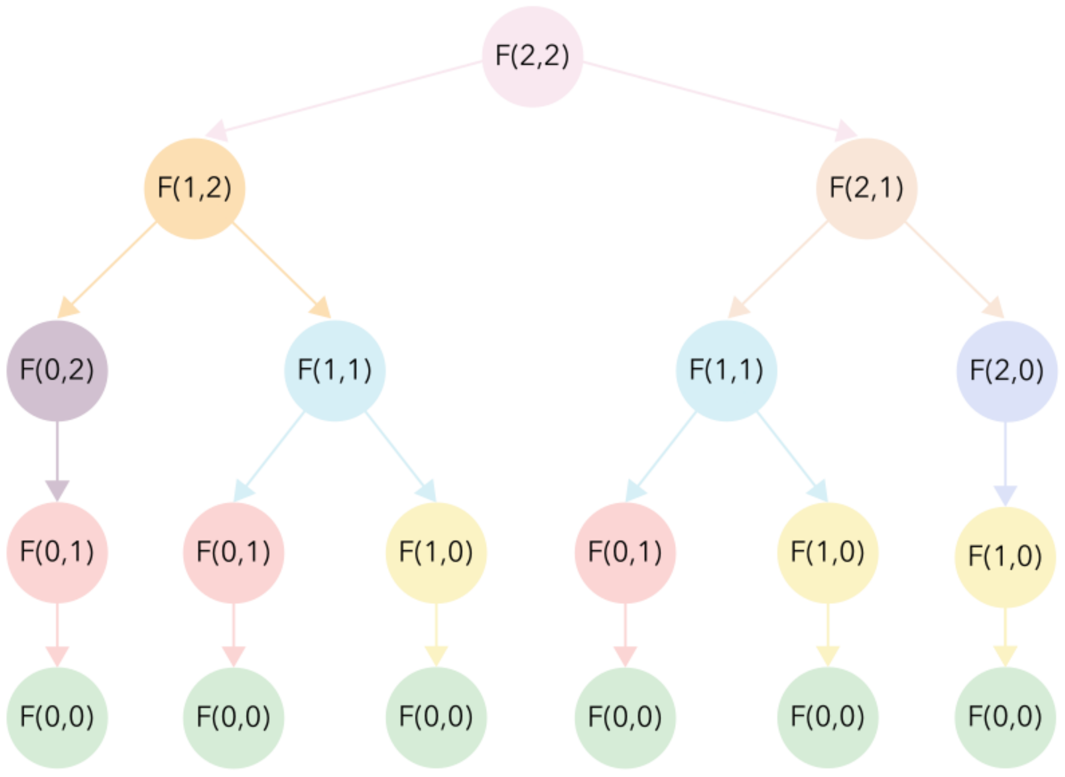

Figure 9. The recursive tree for Unique Paths.

So, the time complexity of the plain recursive solution is $O(2^{n+m}) \approx O(2^n)$..

All the recursive solutions above lead to exponential time complexity due to overlapping sub-problems.

Memoization

How to optimize the recursive solutions above

A recursive solution has overlapping sub-problems that are being calculated several times. To avoid repetitive calculations, we need to store already calculated solutions for sub-problems. Next time, we will also need to use stored values instead of a recalculation. This technique is called **memoization**.

How to derive keys for memoization

To understand how sub-problem solutions are stored, we need to know the keys for the memoization table. According to **The Theory of Dynamic Programming**, we have **state parameters (or state variables)** that determine each state. These parameters transform each state.

"We have a physical system whose state at any time is determined by a set of quantities which we call state parameters, or state variables (The Theory of Dynamic Programming, Richard Bellman)."

"These decisions are equivalent to transformations of the state variables, the choice of a decision being identical with the choice of a transformation. The outcome of the preceding decisions is to be used to guide the choice of future ones, with the purpose of the whole process of maximizing some function of the parameters describing the final state (The Theory of Dynamic Programming, Richard Bellman)."

The statements above give an idea to use state parameters to store information about a particular state's decision.

memo defines our storage for sub-problem solutions.

Fibonacci Sequence

Let's have a look at the recursive solution again.

```
int Fib(int n) {
    if (n == 0) {
        return 0; // base case
    }
    if (n == 1) {
        return 1; // base case
    }
    return Fib(n-1) + Fib(n-2);
}
```

For the Fibonacci Sequence, each state $Fib(int\ n)$ has the term number n as a parameter, transforming each state. Based on this information, we can use the parameter n as the key for the memoization table.

```
int Fib(int n) {
    if (n == 0) {
        return 0; // base case
    }
    if (n == 1) {
        return 1; // base case
    }
    if (memo[n] != -1) { // check if a state was already calculated
```

```
        return memo[n];
    }
    memo[n] = Fib(n-1) + Fib(n-2); // store calculations
    return memo[n];
}
```

Coin Change

The parameter $target$ transforms each state, and we can use it as the key for the memoization table.

```
int CoinChange(int target, vector<int> &coins) {
    if (target == 0) {
        return 0; // base case
    }

    int result = inf;

    for (int i = 0; i < coins.size(); ++i) {
        if (coins[i] <= target) { // check validity of a sub-problem
             result = min(result, CoinChange(target - coins[i], coins) + 1);
         }
     }

     return result;
}
```

Consequently,

```
int CoinChange(int target, vector<int> &coins) {
    if (target == 0) {
        return 0; // base case
    }

    if (memo[target] != -1) { // check if a state was already calculated
        return memo[target];
    }

    int result = inf;

    for (int i = 0; i < coins.size(); ++i) {
         if (coins[i] <= target) { // check validity of a sub-problem
             result = min(result, CoinChange(target - coins[i], coins) + 1);
         }
     }

    memo[target] = result;

    return result;
}
```

Unique Paths

There are two state parameters x and y, which we can use as the keys for the memoization table.

```
int UniquePaths(int x, int y) {
    if (x == 0 && y == 0) {
      return 1; // base case
    }
    if (x < 0 || y < 0) {
        return 0; // not valid sub-problem
    }
    return UniquePaths(x-1, y) + UniquePaths(x, y-1);
}
```

The solution with memoization.

```
int UniquePaths(int x, int y) {
    if (x == 0 && y == 0) {
        return 1; // base case
    }
    if (x < 0 || y < 0) {
        return 0; // not valid sub-problem
    }
    if (memo[x][y] != -1) { // check if a state was already calculated
        return memo[x][y];
    }
    memo[x][y] = UniquePaths(x-1, y) + UniquePaths(x, y-1);
    return memo[x][y];
}
```

In the case above, we define a 2-dimensional array to store sub-state information.

How to calculate the time complexity of a recursive solution with memoization

To calculate the time-complexity of a recursive solution with memoization, try to answer the following questions:

- How many states do we have in total? (N)?
- How many times does a function call itself? (t)?

The only difference is the fact that memoization will help us to avoid repetitive calculations for sub-problems. As a result, the time complexity will be $O(Nt)$.

Fibonacci Sequence

```cpp
int Fib(int n) {

    /*base cases*/

    if (memo[n] != -1) {
        return memo[n]; // check if a state was already calculated
    }
    memo[n] = Fib(n-1) + Fib(n-2); // store calculations
    return memo[n];
}
```

The function is being called two times:

- $F(n-1)$
- $F(n-2)$

We have a total number of n states. The time complexity of the recursive solution with memoization is $O(n)$.

Coin Change

```cpp
int CoinChange(int target, vector<int> &coins) {

    /*base cases*/

    if (memo[target] != -1) {
        return memo[target];
    }

    int result = inf;

    for (int i = 0; i < coins.size(); ++i) {
        if (coins[i] <= target) { // check validity of a sub-problem
            result = min(result, CoinChange(target - coins[i], coins) + 1);
        }
    }
    memo[target] = result;

    return result;
}
```

The function is being called n times:

- $F(target\ -\ coin\ 1)$
- $F(target\ -\ coin\ 2)$
- $F(target\ -\ coin\ 3)$
- ...
- $F(target\ -\ coin\ n)$

where n is the number of coins.

Here we have a total number of $target\ amount$ states. The time complexity of the recursive solution with memoization is $O(target\ amount\ \times\ n)$.

Unique Paths

```
int UniquePaths(int x, int y) {
    /*base cases*/

   if (memo[x][y] != -1) {
        return memo[x][y];
   }
   memo[x][y] = UniquePaths(x-1, y) + UniquePaths(x, y-1);
   return memo[x][y];
}
```

The function is being called two times:

- $F(x-1, y)$
- $F(x, y-1)$

We have a total of nm states. The time complexity of the recursive solution with memoization is $O(nm)$.

Bottom-Up

The **bottom-up** approach solves a problem iteratively, going from a base case towards a target state. The bottom-up method finds an optimal solution for each sub-problem and stores a result for each sub-problem for later use. There are two ways of solving problems iteratively: backward style and forward style.

Backward Style

In the **backward style** iteration, we calculate a solution for the current state based on solutions of the previous states.

$$f(i) = f(i-1) + f(i-2)$$

Fibonacci Sequence

```
table[i] = table[i-1] + table[i-2]
```

Forward Style

In the **forward style** iteration, we calculate a solution for the next state based on solutions of the current state.

$$f(i+1) = f(i) + f(i-1)$$

Fibonacci Sequence

```
table[i+1] = table[i] + table[i-1]
```

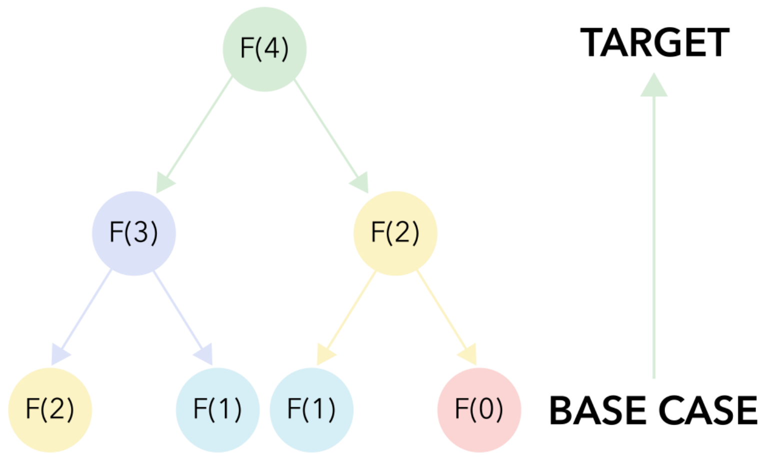

Figure 10. The bottom-up approach.

How to come up with an iterative solution

Perform the following steps to understand how to come up with an iterative solution.

table refers to a table to store sub-problem solutions.

Step 1. Base Case
Find a base case for a given problem.

The base case can be a state we already know or can calculate the answer.

Fibonacci Sequence

We already know that the first two Fibonacci numbers are 0 and 1 respectively.

$$table(0) = 0$$

$$table(1) = 1$$

Coin Change

We don't need any coins to make the total amount of 0.

$$table(0) = 0$$

Unique Paths

We are using zero-indexed positions, and we have only one path to reach $(0, 0)$ position.

$$table(0,0) = 1$$

Step 2. Move Toward a Target State
Starting from a base case, we move toward a target state.

Fibonacci Sequence

$$table(0) = 0$$

$$table(1) = 1$$

$$table(2) = \dots$$

$$\dots$$

$$table(n) = \dots$$

Coin Change

$$table(0) = 0$$

$$table(1) = 1$$

$$table(2) = \ldots$$

$$\ldots$$

$$table(target) = \ldots$$

Unique Paths

$$table(0,0) = 1$$

$$table(0,1) = \ldots$$

$$table(0,2) = \ldots$$

$$\ldots$$

$$table(1,1) = \ldots$$

$$\ldots$$

$$table(x,y) = \ldots$$

Step 3. Iterate Over Options
For every step (state), we have several options (ways) to choose to come to the current state. We need to iterate all possible options to find the most optimal solution for the current state.

Fibonacci Sequence

$$table(n) \rightarrow \begin{matrix} table(n-1) \\ table(n-2) \end{matrix}$$

To obtain n^{th} term there are two options to come from to the current state:

- the last term $n-1$
- one before the last term $n-2$

Coin Change

$$table(target) \rightarrow \begin{matrix} table(target\ -\ coin\ with\ denomination\ 5) \\ table(target\ -\ coin\ with\ denomination\ 3) \\ table(target\ -\ coin\ with\ denomination\ 1) \end{matrix}$$

To make the $target\ amount$, there are three options to come from to the current state:

- using a coin with denomination 5
- using a coin with denomination 3
- using a coin with denomination 1

We are starting from the base case to apply a coin with denomination K check if the current target is greater or equal to the coin denomination.

Unique Paths

$$table(x, y) \rightarrow \begin{matrix} table(x-1, y) \\ table(x, y-1) \end{matrix}$$

To reach the target, there are two options (directions) to come from to the current state:

- from top $x - 1$
- from left $y - 1$

Step 4. Choose an Optimal Solution
We find an optimal solution (maximum, minimum, shortest, longest, etc.) without the current state and add a value for the current state.

Fibonacci Sequence

$$table(n) \ = \ table(n-1) \ + \ table(n-2)$$

Sum up two previous values.

Coin Change

$$table(target) = \ min \begin{pmatrix} table(target \ - \ coin\ with\ denomination\ 5) \\ table(target \ - \ coin\ with\ denomination\ 3) \\ table(target \ - \ coin\ with\ denomination\ 1) \end{pmatrix} + \ 1$$

Find an optimal solution without the current state and add one coin for the current state.

Unique Paths

$$table(x, y) \ = \ table(x-1, y) \ + \ table(x, y-1)$$

Sum up both options to find a total path.

Step 5. Return Result
After reaching a target state, return an answer for the target state.

Fibonacci Sequence

$return\ table(n)$

Coin Change

$return\ table(target)$

Unique Paths

$return\ table(x, y)$

Applying the methods above to solve examples iteratively

Iterative solution for the Fibonacci Sequence

```cpp
int Fib(int n) {
    vector<int> table(n+1);

   table[0] = 0; // step 1
   table[1] = 1; // step 1

   for (int i = 2; i <= n; ++i) { // step 2
       int last = table[i-1]; // step 3
       int beforeLast = table[i-2]; // step 3
       table[i] = last + beforeLast; // step 4
   }

   return table[n]; // step 5
 }
```

Iterative solution for Coin Change Problem

```cpp
int CoinChange(int target, vector<int> &coins) {
    vector<int> table(target+1, inf);

    table[0] = 0; // step 1

    for (int i = 1; i <= target; ++i) { // step 2
        for (int j = 0; j < coins.size(); ++j) { // step 3
            if (coins[j] <= i) {
                table[i] = min(table[i], table[i - coins[j]] + 1); // step 4
            }
        }
    }

    return table[target] == inf ? -1 : table[target]; // step 5
}
```

Iterative solution for Unique Paths

```
int UniquePaths(int x, int y) {
    int table[x][y];

    table[0][0] = 1; // step 1

    for (int i = 0; i < x; ++i) { // step 2
        for (int j = 0; j < y; ++j) { // step 2
            if (i > 0 && j > 0) {
                table[i][j] = table[i-1][j] + table[i][j-1]; // step 3 & 4
            } else if (i > 0) {
                table[i][j] = table[i-1][j]; // step 3 & 4
            } else if (j > 0) {
                table[i][j] = table[i][j-1]; // step 3 & 4
            }
        }
    }

    return table[x-1][y-1]; // // step 5
}
```

How to calculate the time complexity of the iterative solution

To calculate the time complexity of the iterative solution, we need to count how many operations we are performing inside the function.

Fibonacci Sequence

To find n^{th} Fibonacci number, we need to iterate over all Fibonacci numbers till n^{th} term. Inside the single loop, we are performing constant operations $O(1)$. Time complexity is $O(n)$.

```
int Fib(int n) {

    /*code*/

    for (int i = 2; i <= n; ++i) {
        table[i] = table[i-1] + table[i-2];
    }

    return table[n];
}
```

Coin Change

To find the minimum number of coins to make $target\ amount$, we need to iterate over all options for every target. Inside the nested loop, we are performing constant operations $O(1)$. The time complexity is $O(target\ amount\ *\ n)$, where n is the number of coins.

```
int CoinChange(int target, vector<int> &coins) {

    /* code */

    for (int i = 1; i <= target; ++i) {
        for (int j = 0; j < coins.size(); ++j) {
            if (coins[j] <= i) {
                table[i] = min(table[i], table[i - coins[j]] + 1);
            }
        }
    }

    return table[target] == inf ? -1 : table[target];
}
```

Unique Paths

To reach the target position, we need to iterate over all positions and sum up unique paths. The time complexity is $O(nm)$.

```
int UniquePaths(int x, int y) {

    /* code */

    for (int i = 0; i < x; ++i) { // iterate over rows
        for (int j = 0; j < y; ++j) { // iterate over columns
            if (i > 0 && j > 0) {
                table[i][j] = table[i-1][j] + table[i][j-1]; // choose options
            } else if (i > 0) {
                table[i][j] = table[i-1][j]; // choose options
            } else if (j > 0) {
                table[i][j] = table[i][j-1]; // choose options
            }
        }
    }

    return table[x-1][y-1]; // return result
}
```

Top-Down vs. Bottom-Up

The **top-down (recursive)** solution is easy to develop, though it needs extra space for a recursive call stack. The **bottom-up (iterative)** solution is not intuitive to come up with, but it avoids extra space.

From Top-Down to Bottom-Up

To derive an iterative solution from a recursive solution, we need to discover common patterns between these two methods.

Step 1. Base Case
Reuse base cases.

Both methods have base cases to start and to finish operations. After finding a recursive solution, we can use base cases of a recursive solution for an iterative solution.

```
int topDown(int n) {
    if (base case) {
        return result; // base case
    }
}
```

```
int bottomUp(int n) {
    table[base case] = result;
}
```

Fibonacci Sequence

```
int topDown(int n) {
    if (n == 0) {
        return 0; // base case
    }
    if (n == 1) {
        return 1; // base case
    }

  ...
 }
```

```
int bottomUp(int n) {
    ...

    table[0] = 0; // base case
    table[1] = 1; // base case

    ...
}
```

Coin Change

```
int topDown(int n) {
    if (target == 0) {
        return 0; // base case
    }

    ...
}
```

```
int bottomUp(int n) {
    ...

    table[0] = 0; // base case

    ...
}
```

Unique Paths

```
int topDown(int n) {
    if (x == 0 && y == 0) {
        return 1; // base case
    }
    if (x < 0 || y < 0) {
        return 0; // not valid sub-problem
    }

    ...
}
```

```
int bottomUp(int n) {
    ...

    table[0][0] = 1; // base case

    ...
}
```

Step 2. Table to Store Sub-State Solutions
Reuse the memoization table from a recursive solution.

In a recursive solution, there is a memoization table to store sub-state solutions. We can use the same table for an iterative solution as well to store sub-state solutions.

Fibonacci Sequence

```cpp
int topDown(int n, vector<int>& memo) {
    ...

    if (memo[n] != -1) {
        return memo[n]; // check if the state was already calculated
    }

    memo[n] = /* result */

    return result;
}
```

```cpp
int bottomUp(int n) {
    ...

    for (...) {
        table[n] = /* result */
    }

    return table[n];
}
```

Coin Change

```cpp
int topDown(int target, vector<int> &coins, vector<int>& memo) {
    ...

    if (memo[target] != -1) {
        return memo[target];
    }

    ...

    memo[target] = /* result */

    return result;
}
```

```
int bottomUp(int target, vector<int> &coins) {
    ...

    for (...) {
        for (...) {
            if (...) {
                table[target] = /* result */
            }
        }
    }

    return table[target] == inf ? -1 : table[target];
}
```

Unique Paths

```
int topDown(int x, int y, vector<vector<int>>& memo) {
    ...

    if (memo[x][y] != -1) {
        return memo[x][y];
    }

    memo[x][y] = /* result */

    return result;
}
```

```
int bottomUp(int x, int y) {
    int table[x][y];

    ...

    for (...) {
        for (...) {
            table[i][j] = /* result */
        }
    }

    return table[x-1][y-1];
}
```

Step 3. Iterative Function Calls
Write iterative function calls.

Now it's time to find out how to replace recursive calls with iterative calls. Let's recall the statement from the paper.

> "We have a physical system whose state at any time is determined by a set of quantities which we call state parameters, or state variables (The Theory of Dynamic Programming, Richard Bellman)."

Consequently, we can use state parameters used in recursive calls for iterative calls as well.

Fibonacci Sequence

n is a state parameter transforming each state.

```cpp
int topDown(int n, vector<int>& memo) {
    ...

    int result = Fib(n-1) + Fib(n-2); // n - transforming state parameter

    memo[n] = result;

    return result;
}
```

```cpp
int bottomUp(int n) {
    ...

    for (int i = 2; i <= n; ++i) { // n - transforming state parameter
        table[i] = /* result */
    }

    return table[n];
}
```

Coin Change

target is a state parameter transforming each state.

```cpp
int topDown(int target, vector<int> &coins, vector<int>& memo) {
    ...

    for (...) {
        if (coin <= target) {
            result = CoinChange(target - coin, coins); // target - transforming state parameter
        }
    }

    memo[target] = result;

    return result;
}
```

```cpp
int bottomUp(int target, vector<int> &coins) {
    ...

    for (int i = 1; i <= target; ++i) { // target - transforming state parameter
        for (...) {
            if (...) {
                table[i] = /* result */
            }
        }
    }

    return table[target] == inf ? -1 : table[target];
}
```

Unique Paths

x and y are state parameters transforming each state.

```cpp
int topDown(int x, int y, vector<vector<int>>& memo) {
    ...

    int result = UniquePaths(x-1, y) + UniquePaths(x, y-1); // x and y - transforming state parameters

    memo[x][y] = result;

    return memo[x][y];
}
```

```
int bottomUp(int x, int y) {
    ...

    for (int i = 0; i < x; ++i) { // x - transforming state parameter
        for (int j = 0; j < y; ++j) { // y - transforming state parameter
            table[i][j] = /* result */
        }
    }

    return table[x-1][y-1];
}
```

Step 4. Iterate Over Options
Both top-down and bottom-up approaches require iterating over all possible options to find the most optimal solution to reach a target state.

Fibonacci Sequence

$$table(n) \rightarrow \begin{matrix} table(n-1) \\ table(n-2) \end{matrix}$$

$table$ refers to a table to store solutions for sub-problems.

To obtain n^{th} term, there are two options to come from to the current state:

- the last term
- one before the last term

```
int topDown(int n, vector<int>& memo) {
    ...

    int result = Fib(n-1) + Fib(n-2); // Iterate over all options.

    memo[n] = result;

    return result;
}
```

```
int bottomUp(int n) {
    ...

    for (int i = 2; i <= n; ++i) {
        table[i] = table[i-1] + table[i-2];. // Iterate over all options
    }

    return table[n];
}
```

Coin Change

$$table(target) \rightarrow \begin{array}{l} table(target - coin\ with\ denomination\ 5) \\ table(target - coin\ with\ denomination\ 3) \\ table(target - coin\ with\ denomination\ 1) \end{array}$$

To make the target amount, there are three options to come from to the current state:

- using the coin with denomination 5
- using the coin with denomination 3
- using the coin with denomination 1

```
int topDown(int target, vector<int>& coins, vector<int>& memo) {
    ...

    for (int i = 0; i < coins.size(); ++i) { // Iterate over all options.
        if (coins[i] <= target) {
            result = min(result, CoinChange(target - coins[i], coins) + 1);
        }
    }

    memo[target] = result;

    return result;
}
```

```
int bottomUp(int target, vector<int> &coins) {
    ...

    for (int i = 1; i <= target; ++i) {
        for (int j = 0; j < coins.size(); ++j) { // Iterate over all options.
            if (coins[j] <= i) {
                table[i] = min(table[i], table[i - coins[j]] + 1);
            }
        }
    }
```

```cpp

    return table[target] == inf ? -1 : table[target];
}
```

Unique Paths

$$table(x, y) \rightarrow \begin{matrix} table(x-1, y) \\ table(x, y-1) \end{matrix}$$

To reach the target, there are two options (directions) to come from to the current state:

- from top $x - 1$
- from left $y - 1$

```cpp
int topDown(int x, int y, vector<vector<int>>& memo) {
    ...

    memo[x][y] = UniquePaths(x-1, y) + UniquePaths(x, y-1); // Iterate over all options and reduce the problem with the current option.

    return memo[x][y];
}
```

```cpp
int bottomUp(int x, int y) {
    ...

    for (int i = 0; i < x; ++i) { // x - transforming state parameter
        for (int j = 0; j < y; ++j) { // y - transforming state parameter
            if (i > 0 && j > 0) {
                table[i][j] = table[i-1][j] + table[i][j-1]; // Iterate over all options.
            } else if (i > 0) {
                table[i][j] = table[i-1][j]; // Iterate over all options.
            } else if (j > 0) {
                table[i][j] = table[i][j-1]; // Iterate over all options.
            }
        }
    }

    return table[x-1][y-1];
}
```

Step 5. Choose an Optimal Solution
Find an optimal solution (maximum, minimum, shortest, longest, etc.) among all states and add a value for the current state.

For every state, we need to choose the most optimal sub-state solution and add the value for the current state.

Fibonacci Sequence

$$table(n) \; = \; table(n-1) \; + \; table(n-2)$$

Sum up two previous values.

```
int topDown(int n, vector<int>& memo) {
    ...

    int result = Fib(n-1) + Fib(n-2); // Sum up two previous values

    memo[n] = result;

    return result;
}
```

```
int bottomUp(int n) {
    ...

    for (int i = 2; i <= n; ++i) {
        table[i] = table[i-1] + table[i-2];. // Sum up two previous values
    }

    return table[n];
}
```

Coin Change

$$table(target) = min\begin{pmatrix} table(target \; - \; coin \; with \; denomination \; 5) \\ table(target \; - \; coin \; with \; denomination \; 3) \\ table(target \; - \; coin \; with \; denomination \; 1) \end{pmatrix} + \; 1$$

Find an optimal solution without the current state and add one coin for the current state.

```
int topDown(int target, vector<int>& coins, vector<int>& memo) {
    ...
```

```cpp

    for (int i = 0; i < coins.size(); ++i) { // Iterate over all options.
        if (coins[i] <= target) {
            result = min(result, CoinChange(target - coins[i], coins) + 1); // Find an optimal solution
        }
    }

    memo[target] = result;

    return result;
}
```

```cpp
int bottomUp(int target, vector<int> &coins) {
    ...

    for (int i = 1; i <= target; ++i) {
        for (int j = 0; j < coins.size(); ++j) { // Iterate over all options.
            if (coins[j] <= i) {
                table[i] = min(table[i], table[i - coins[j]] + 1); // Find an optimal solution
            }
        }
    }

    return table[target] == inf ? -1 : table[target];
}
```

Unique Paths

$$table(x,y) = table(x-1,y) + table(x,y-1)$$

Sum up both options to find the total number of paths.

```cpp
int topDown(int x, int y, vector<vector<int>>& memo) {
    ...

    memo[x][y] = UniquePaths(x-1, y) + UniquePaths(x, y-1); // Sum up both options to find a total path.

    return memo[x][y];
}
```

```cpp
int bottomUp(int x, int y) {
    ...

    for (int i = 0; i < x; ++i) { // x - transforming state parameter
        for (int j = 0; j < y; ++j) { // y - transforming state parameter
            if (i > 0 && j > 0) {
                table[i][j] = table[i-1][j] + table[i][j-1]; // Sum up both options to find a total path.
            } else if (i > 0) {
                table[i][j] = table[i-1][j]; // Sum up both options to find a total path.
            } else if (j > 0) {
                table[i][j] = table[i][j-1]; // Sum up both options to find a total path.
            }
        }
    }

    return table[x-1][y-1];
}
```

Step 6. Return Result
After reaching a target state, return an answer for the target state.

Fibonacci Sequence

`return table[n]`

Coin Change

`return table[target]`

Unique Paths

`return table[x-1][y-1]`

Practicing Common Dynamic Programming Problems

Minimum Cost Climbing Stairs

Find the minimum cost required to reach the top stair from the bottom stair, climbing either 1 or 2 stairs each time. To climb, you need to pay the cost.

Intuition

For every stair, we choose an option that leads to the minimum cost among all possible options before the current stair, and we add the cost for the current stair.

Top-Down

Step 1. Iterate Over Options
When we solve a problem recursively, for every sub-problem (state), we have different options (directions) to choose to reach our target. We need to iterate over all possible options to find the most optimal solution.

To reach the top stair, there are two options to choose from:

- climb 1 stair
- climb 2 stairs

$$F(current\ stair) \rightarrow \begin{matrix} F(climb\ 1\ stair) \\ F(climb\ 2\ stairs) \end{matrix}$$

Step 2. Reduce Target
Choosing every option, we reduce a target to a particular value, and we call a recursive function again with the new updated target.

$$F(n) \rightarrow \begin{matrix} F(n-1) \\ F(n-2) \end{matrix}$$

- Climbing 1 stair reduces the problem to find the minimum cost required to reach $n - 1^{th}$ stair.
- Climbing 2 stairs reduces the problem to find the minimum cost required to reach $n - 2^{th}$ stair.

$$result \rightarrow \begin{matrix} minCost(n-1, cost, memo) \\ minCost(n-2, cost, memo) \end{matrix}$$

Step 3. Choose Optimal Solution

After coming from recursive calls, we choose an optimal solution among these calls and add the value for the current state.

$$F(n) = min\begin{pmatrix}F(n-1)\\F(n-2)\end{pmatrix} + cost$$

We choose the minimum cost among all sub-problem solutions before the current stair, and we add the cost for the current stair.

$$result = min\begin{pmatrix}minCost(n-1, cost, memo)\\minCost(n-2, cost, memo)\end{pmatrix} + cost[n]$$

Step 4. Base Case

When we reach a base case, we need to quit a recursive function. The base case can be a base state or case when a sub-problem is no longer valid (in this case, when it returns the worst result).

If you start to climb from 0 floor, you will pay $cost[0]$, and if you start to climb from 1st floor, you will pay $cost[1]$.

```
if (n == 0) {
    return cost[0]; // base case
}
if (n == 1) {
    return cost[1]; // base case
}
```

Step 5. Memoization

We use transforming state variables for memorization.

n is a state variable transforming each state. We can use it for memoization.

```
if (memo[n] != -1) {
    return memo[n];
}
```

Step 6. Return Result

Return a result of a recursive function.

```
return result
```

Solution

```cpp
int topDown(int n, vector<int>& cost, vector<int>& memo) {
    if (n == 0) {
        return cost[n]; // base case
    }
    if (n == 1) {
        return cost[n]; // base case
    }

    if (memo[n] != -1) {
        return memo[n];
    }

    int result = min(minCost(n-1, cost, memo), minCost(n-2, cost, memo)) + (n == cost.size() ? 0 : cost[n]);

    return memo[n] = result;
}
```

Analysis

Time-Complexity: $O(n)$

We are calling the function two times and have a total of n states, where n is the number of stairs.

Space-Complexity: $O(n)$

We are using n space to store sub-state solutions.

Bottom-Up

Step 1. Base Case
Find a base case for a given problem.

Base cases are $n = 0$ and $n = 1$ because we can start climbing from stairs 0 and 1 and know the costs for these stairs.

```cpp
dp[0] = cost[0]; // base case
dp[1] = cost[1]; // base case
```

Step 2. Move Toward a Target State
Starting from a base case, we move toward a target state.

We already know the solution for the base cases. To find solutions for other stairs, we iterate till n.

```
dp[0] = cost[0]; // base case
dp[1] = cost[1]; // base case

for (int i = 2; i <= n; ++i) { // move towards n
    /* code */
}
```

Step 3. Iterate Over Options
For every step (state), we have several options (ways) to choose to come to the current state. We need to iterate all possible options to find the most optimal solution for the current state.

We can come to the current stair either by climbing 1 or 2 stairs.

```
for (int i = 2; i <= n; ++i) {
    dp[i] <- dp[i-1], dp[i-2]; // iterate over options
}
```

Step 4. Choose an Optimal Solution
We find an optimal solution (maximum, minimum, shortest, longest, etc.) without the current state and add a value for the current state.

We choose an option with the minimum cost before the current stair, and we add the cost for the current stair.

```
for (int i = 2; i <= n; ++i) {
    dp[i] = min(dp[i-1], dp[i-2]) + cost[i];
}
```

Step 5. Return Result
After reaching a target state, return an answer for the target state.

We return the answer for the target stair.

`return dp[n]`

Solution

```cpp
int bottomUp(vector<int>& cost) {
    int n = (int)cost.size();

    vector<int> dp(n+1);

    dp[0] = cost[0]; // base case
    dp[1] = cost[1]; // base case

    for (int i = 2; i <= n; ++i) {
        dp[i] = min(dp[i-1], dp[i-2]) + (i == n ? 0 : cost[i]);
    }

    return dp[n];
}
```

Analysis

Time-Complexity: $O(n)$

Inside the single loop, we are performing constant $O(1)$ operations n times, where n is the number of stairs.

Space-Complexity: $O(n)$

We are using n space to store sub-state solutions.

Minimum Path Sum

Given a 2D grid with non-negative numbers, find the path from the top-left corner to the bottom-right corner with the minimum sum of all numbers along the path.

Intuition

For every position, we choose the path with the minimum sum among all paths before the current position, and we add the value for the current position.

Top-Down

Step 1. Iterate Over Options
When we solve a problem recursively, for every sub-problem (state), we have different options (directions) to choose to reach our target. We need to iterate over all possible options to find the most optimal solution.

To reach the target position, there are two options to choose from:

- move down $i + 1$
- move right $j + 1$

$$F(target\ position) \rightarrow \begin{matrix} F(move\ down) \\ F(move\ right) \end{matrix}$$

Step 2. Reduce Target
Choosing every option, we reduce a target to a particular value, and we call a recursive function again with the new updated target.

$$F(target\ position) \rightarrow \begin{matrix} F(i+1, j) \\ F(i, j+1) \end{matrix}$$

- Moving down $i + 1$ makes us closer to the target position in terms of a row.
- Moving right $j + 1$ makes us closer to the target position in terms of a column.

$$result \rightarrow \begin{matrix} pathSum(i+1, j, grid, memo) \\ pathSum(i, j+1, grid, memo) \end{matrix}$$

Step 3. Choose Optimal Solution

After coming from recursive calls, we choose an optimal solution among these calls and add the value for the current state.

$$F(n) = min\begin{pmatrix}F(i+1,j)\\F(i,j+1)\end{pmatrix} + value$$

We choose the minimum sum among sub-problem solutions, and we add the value for the current position.

$$result = min\begin{pmatrix}pathSum(i+1,j,grid,memo)\\pathSum(i,j+1,grid,memo)\end{pmatrix} + grid[i][j]$$

Step 4. Base Case

When we reach a base case, we need to quit a recursive function. The base case can be a base state or case when a sub-problem is no longer valid (in this case, when it returns the worst result).

If the position reaches the target position, there is no need to move further; we return the value for the target position. If the position is out of bounds of the grid, then the problem becomes invalid, we return the worst result.

```
// reach the target
if (i == grid.size()-1 && j == grid[0].size()-1) {
    return grid[i][j];
}

// invalid position
if (i < 0 || i >= grid.size() || j < 0 || j >= grid[0].size()) {
    return INT_MAX;
}
```

Step 5. Memoization

We use transforming state variables for memorization.

Indices i and j are state variables transforming each state. We can use them for memoization.

```
if (memo[i][j] != -1) {
    return memo[i][j];
}
```

Step 6. Return Result

Return a result of a recursive function.

```
return memo[i][j]
```

Solution

```
int pathSum(int i, int j, vector<vector<int>>& grid, vector<vector<int>>& memo) {
    // reach the target
    if (i == grid.size()-1 && j == grid[0].size()-1) {
        return grid[i][j];
    }

    // invalid position
    if (i < 0 || i >= grid.size() || j < 0 || j >= grid[0].size()) {
        return INT_MAX;
    }

    if (memo[i][j] != -1) {
        return memo[i][j];
    }

    int result = min(pathSum(i+1, j, grid, memo), pathSum(i, j+1, grid, memo)) +
grid[i][j];

    memo[i][j] = result;
    return result;
}
```

Analysis

Time-Complexity: $O(nm)$

We are calling the function two times and have a total of nm states, where n is the number of rows and m is the number of columns.

Space-Complexity: $O(nm)$

We are using nm space to store sub-state solutions.

Bottom-Up

Step 1. Base Case
Find a base case for a given problem.

For base cases, we can calculate the values for the first row ignoring other rows. This means that we have only one option to choose from, the same row previous column. Similarly, we can calculate the values for the first column ignoring other columns. In this case, we can choose only the same column previous row.

```
for (int j = 1; j < m; ++j) {
    grid[0][j] += grid[0][j-1];
}

for (int i = 1; i < n; ++i) {
    grid[i][0] += grid[i-1][0];
}
```

Step 2. Move Toward a Target State
Starting from a base case, we move toward a target state.

We iterate over all positions moving towards the target position $(n-1, m-1)$.

```
for (int i = 1; i < n; ++i) {
    for (int j = 1; j < m; ++j) {
        /* code */
    }
}
```

Step 3. Iterate Over Options
For every step (state), we have several options (ways) to choose to come to the current state. We need to iterate all possible options to find the most optimal solution for the current state.

We can come to the current position from the top $(i-1, j)$ and the left $(i, j-1)$ side positions.

```
for (int i = 1; i < n; ++i) {
    for (int j = 1; j < m; ++j) {
        grid[i][j] <- grid[i-1][j], grid[i][j-1]
    }
}
```

Step 4. Choose an Optimal Solution
We find an optimal solution (maximum, minimum, shortest, longest, etc.) without the current state and add a value for the current state.

We choose an option with the minimum sum and add the value for the current position.

```
for (int i = 1; i < n; ++i) {
    for (int j = 1; j < m; ++j) {
        grid[i][j] = min(grid[i-1][j], grid[i][j-1]) + grid[i][j];
    }
}
```

Step 5. Return Result
After reaching a target state, return an answer for the target state.

We return the answer for the target position.

`return grid[n-1][m-1]`

Solution

```
int minPathSum(vector<vector<int>>& grid) {
    if (grid.size() == 0) {
        return 0;
    }

    int n = (int)grid.size();
    int m = (int)grid[0].size();

    for (int j = 1; j < m; ++j) {
        grid[0][j] += grid[0][j-1];
    }

    for (int i = 1; i < n; ++i) {
        grid[i][0] += grid[i-1][0];
    }

    for (int i = 1; i < n; ++i) {
        for (int j = 1; j < m; ++j) {
            grid[i][j] = min(grid[i-1][j], grid[i][j-1]) + grid[i][j];
        }
    }

    return grid[n-1][m-1];
}
```

Analysis

Time-Complexity: $O(nm)$

Inside the nested loop, we are performing constant $O(1)$ operations nm times, where n is the number of rows and m is the number of columns.

Space-Complexity: $O(nm)$

We are using nm space to store sub-state solutions.

Climbing Stairs

Find the number of distinct ways to reach the top stair from the bottom stair, climbing either 1 or 2 stairs each time.

Intuition

For every stair, we sum all possible ways to reach the current stair. The solution for the top stair will be the answer.

Top-Down

Step 1. Iterate Over Options
When we solve a problem recursively, for every sub-problem (state), we have different options (directions) to choose to reach our target. We need to iterate over all possible options to find the most optimal solution.

To reach the top stair, there are two options to choose from:

- climb 1 stair
- climb 2 stairs

$$F(current\ stair) \rightarrow \begin{matrix} F(climb\ 1\ stair) \\ F(climb\ 2\ stairs) \end{matrix}$$

Step 2. Reduce Target
Choosing every option, we reduce a target to a particular value, and we call a recursive function again with the new updated target.

$$F(n) \rightarrow \begin{matrix} F(n-1) \\ F(n-2) \end{matrix}$$

- Climbing 1 stair reduces the problem to find the number of distinct ways to reach the $n-1^{th}$ stair.
- Climbing 2 stairs reduces the problem to find the number of distinct ways to reach the $n-2^{th}$ stair.

$$result \rightarrow \begin{matrix} climbStairs(n-1, cost, memo) \\ climbStairs(n-2, cost, memo) \end{matrix}$$

Step 3. Choose Optimal Solution
After coming from recursive calls, we choose an optimal solution among these calls and add the value for the current state.

We sum up both options to get a total number of distinct ways.

$$F(n) = F(n-1) + F(n-2)$$

$$result = climbStairs(n-1, memo) + climbStairs(n-2, memo)$$

Step 4. Base Case
When we reach a base case, we need to quit a recursive function. The base case can be a base state or case when a sub-problem is no longer valid (in this case, when it returns the worst result).

The following scenarios can be base cases:

- If there is 0 floor left to climb, there is only one distinct way.
- If there is 1 floor left to climb, there is only one distinct way to reach the top, to climb 1 stair.

```
if (n < 2) {
    return 1; // base case
}
```

Step 5. Memoization

We use transforming state variables for memorization.

n is a state variable transforming each state. We will use it for memoization.

```
if (memo[n] != -1) {
    return memo[n];
}
```

Step 6. Return Result
Return a result of a recursive function.

```
return memo[n]
```

Solution

```
int climbStairs(int n, vector<int>& memo) {
    if (n < 2) {
        return 1; // base case
    }

    if (memo[n] != -1) {
        return memo[n];
    }

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

    return memo[n];
}
```

Analysis

Time-Complexity: $O(n)$

We are calling the function two times and have a total of n states, where n is the number of stairs.

Space-Complexity: $O(n)$

We are using n space to store sub-state solutions.

Bottom-Up

Step 1. Base Case
Find a base case for a given problem.

The following scenarios can be base cases:

- If there is 0 floor left to climb, there is only one distinct way.
- If there is 1 floor left to climb, there is only one distinct way to reach the top, to climb 1 stair.

```
dp[0] = 1; // base case
dp[1] = 1; // base case
```

Step 2. Move Toward a Target State
Starting from a base case, we move toward a target state.

We already know the answer for the base cases. We iterate over other floors till n^{th} floor to find out solutions for the rest of the floors.

```
dp[0] = 1; // base case
dp[1] = 1; // base case

for(int i = 2; i<=n; i++) { // move towards n
   /* code */
}
```

Step 3. Iterate Over Options
For every step (state), we have several options (ways) to choose to come to the current state. We need to iterate all possible options to find the most optimal solution for the current state.

We can come to the current stair either by making 1 or 2 climbs.

```
for (int i = 2; i<=n; ++i) {
    dp[i] = ...; // iterate over options
}
```

Step 4. Choose an Optimal Solution
We find an optimal solution (maximum, minimum, shortest, longest, etc.) without the current state and add a value for the current state.

We sum up all possible options.

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

Step 5. Return Result
After reaching a target state, return an answer for the target state.

We return the answer for the top stair.

`Return dp[n]`

Solution

```
int bottomUp(int n) {
    if(n < 2) {
        return 1; // early exit
    }

    int dp[n+1];

    dp[0] = 1; // base case
    dp[1] = 1; // base case

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

    return dp[n];
}
```

Analysis

Time-Complexity: $O(n)$

Inside the single loop, we are performing constant $O(1)$ operations n times.

Space-Complexity: $O(n)$

We are using n space to store sub-state solutions.

Note

Some questions point out the number of repetitions. In that case, add one more loop to simulate every repetition.

Burst Balloons

Given balloons with values, after the burst of i^{th} balloon you will get $balloons[i-1] * balloons[i] * balloons[i+1]$ coins. After each burst, balloons $i-1$ and $i+1$ become adjacent. Find the maximum number of coins you can get by bursting balloons.

Intuition

We iterate over every interval, then for every balloon in the interval, we check the possible maximum number of coins we can get if we burst this balloon last.

Top-Down

Step 1. Iterate Over Options
When we solve a problem recursively, for every sub-problem (state), we have different options (directions) to choose to reach our target. We need to iterate over all possible options to find the most optimal solution.

We have several options to burst balloons. For every interval in the list of balloons, we can try to burst balloons at every index last to get the maximum number of coins. Therefore, we need to iterate through indices of balloons.

```
// starting from i to j
for (int k = i; k <= j; ++k) {
        /* code */
}
```

Step 2. Reduce Target
Choosing every option, we reduce a target to a particular value, and we call a recursive function again with the new updated target.

Every time we choose some index to burst a balloon, we split the array of balloons into two parts at that index (excluding the index). Then we call the recursive function again with the new intervals.

$$F(n) \rightarrow \begin{matrix} F(i, k-1) \\ product(k) \\ F(k+1, j) \end{matrix}$$

Choosing index k:

- The first half will be from i till $k-1$.
- The second half will be from $k+1$ till j.

$$result \rightarrow \begin{array}{c} topDown(nums, i, k-1, memo) \\ nums[i-1] * nums[k] * nums[j+1] \\ topDown(nums, k+1, j, memo) \end{array}$$

Step 3. Choose Optimal Solution
After coming from recursive calls, we choose an optimal solution among these calls and add the value for the current state.

$$F(n) = F(i, k-1) + product(k) + F(k+1, j)$$

For every interval, if we decide to burst the balloon in the interval last, we get the best from the left and right sides. Then we add the product of the current balloon with two neighbouring balloons outside the interval. The reason for that is the fact that when we burst the current balloon last, we do not have other unburst balloons in the interval. Remember that each time we burst a balloon, two neighbouring balloons become adjacent.

$$result = topDown(nums, i, k-1, memo) + nums[i-1] * nums[k] * nums[j+1] + topDown(nums, k+1, j, memo)$$

Step 4. Base Case
When we reach a base case, we need to quit a recursive function. The base case can be a base state or case when a sub-problem is no longer valid (in this case, when it returns the worst result).

The following scenarios can be base cases:

- If we can't further divide the interval into two halves $i == j$.
- If the interval is not valid, $i < j$.

```
if (i > j) {
    return 0; // base case
}

if (i == j) {
    return nums[i]; // base case
}
```

Step 5. Memoization

We use transforming state variables for memorization.

Starting index i and ending index j are state variables transforming each state. We can use them for memorization.

```
if (memo[i][j] != -1) {
    return memo[i][j];
}
```

Step 6. Return Result
Return a result of a recursive function.

```
return result
```

Solution

```
int topDown(vector<int>& nums, int i, int j, vector<vector<int>>& memo) {
    if (i > j) {
        return 0; // base case
    }

    if (i == j) {
        return (i-1 >= 0 ? nums[i-1] : 1) * nums[i] * (j+1 < nums.size() ?  nums[j+1] : 
1);
    }

    if (memo[i][j] != -1) {
        return memo[i][j];
    }

    int result = INT_MIN;

    for (int k = i; k <= j; ++k) {
        result = max(result, topDown(nums, i, k-1, memo) + (i-1 >= 0 ? nums[i-1] : 1) * 
nums[k] * (j+1 < nums.size() ?  nums[j+1] : 1) + topDown(nums, k+1, j, memo));
    }

    memo[i][j] = result

    return result;
}
```

Analysis

Time-Complexity: $O(n^3)$

We are calling the function n times and have a total of n^2 states, where n is the number of balloons.

Space-Complexity: $O(n^2)$

We are using n^2 space to store sub-state solutions.

Bottom-Up

Step 1. Base Case
Find a base case for a given problem.

We can calculate a base case for an interval with a length less than or equal to three.

$$nums[0], nums[1], nums[2]$$

$$nums[1], nums[2], nums[3]$$

...

$$nums[n-2], nums[n-1], nums[n]$$

```
int dp[n+1][n+1];

memset(dp, 0, sizeof dp);

for(int i = 0; i < n; i++) {
    dp[i][i] = (i > 0 ? nums[i-1] : 1) * nums[i] * (i < n-1 ? nums[i+1] : 1);
}
```

Step 2. Move Toward a Target State
Starting from a base case, we move toward a target state.

We need to find an answer for the whole interval; therefore, we will be increasing the interval by increasing the length of the interval (considering one more element each time).

```
for(int l = 1; l < n; l++) {
    for(int i = 0; i < n-l; i++) {
        int j = i+l;
        // interval from i to j
    }
}
```

Step 3. Iterate Over Options
For every step (state), we have several options (ways) to choose to come to the current state. We need to iterate all possible options to find the most optimal solution for the current state.

For every interval, we have several options to burst balloons. We can try to burst balloons at every index to get the maximum number of coins. Therefore, we need to iterate through indices of balloons k.

```
for(int l = 1; l < n; l++) {
    for(int i = 0; i < n-l; i++) {
        int j = i+l;
        // interval from i to j
        for(int k = i; k <= j; k++) {
            // try every index
        }
    }
}
```

Step 4. Choose an Optimal Solution
We find an optimal solution (maximum, minimum, shortest, longest, etc.) without the current state and add a value for the current state.

For every interval, we iterate over balloons in the interval. Then we check the maximum number of coins we can get if we burst the current balloon last. If we burst the current balloon last, we take maximum values from the left and right sides and add a product of the current balloon with left and right balloons outside the interval.

```
for(int l = 1; l < n; l++) {
    for(int i = 0; i < n-l; i++) {
        int j = i+l;
        for(int k = i; k <= j; k++) {
            dp[i][j] = max(dp[i][j], dp[i][k-1] + nums[i-1] * nums[k] * nums[j+1] +
dp[k+1][j]);
        }
    }
}
```

Step 5. Return Result
After reaching a target state, return an answer for the target state.

We return the answer for the target state.

`return dp[0][n-1]`

Solution

```
int bottomUp(vector<int>& nums) {
    if(nums.size() == 0) {
        return 0;
    }

    int n = (int)nums.size();

    int dp[n+1][n+1];

    memset(dp, 0, sizeof dp);

    for(int i = 0; i < n; i++) {
        dp[i][i] = (i>0 ? nums[i-1] : 1) * nums[i] * (i<n-1 ? nums[i+1] : 1);
    }

    for(int l = 1; l < n; l++) {
        for(int i = 0; i < n-l; i++) {
            int j = i+l;
            for(int k = i; k <= j; k++) {
                int leftVal = (k>i && k>0) ? dp[i][k-1] : 0;
                int rightVal = (k<j && k<n-1) ? dp[k+1][j] : 0;
                int product = (i>0 ? nums[i-1] : 1)*nums[k]*(j<n-1 ? nums[j+1] : 1);
                dp[i][j] = max(dp[i][j], leftVal + product + rightVal);
            }
        }
    }

    return dp[0][n-1];
}
```

Analysis

Time-Complexity: $O(n^3)$

Inside the nested loop, we are performing constant operations n^3 times, where n is the number of balloons.

Space-Complexity: $O(n^2)$

We are using n^2 space to store sub-state solutions.

Longest Common Subsequence

Given two strings, $string\ 1$ and $string\ 2$, find the length of the longest common subsequence in these strings.

$$\begin{array}{l} i\ -\ the\ index\ of\ string\ 1 \\ j\ -\ the\ index\ of\ string\ 2 \end{array}$$

Top-Down

Step 1. Iterate Over Options
When we solve a problem recursively, for every sub-problem (state), we have different options (directions) to choose to reach our target. We need to iterate over all possible options to find the most optimal solution.

For each state, there are three options to choose from:

- If characters at indices i and j match, move indices for both strings.
- Move the index i for $string\ 1$.
- Move the index j for $string\ 2$.

Step 2. Reduce Target
Choosing every option, we reduce a target to a particular value, and we call a recursive function again with the new updated target.

$$F(i,j) \rightarrow \begin{array}{c} F(i+1,j) \\ F(i+1,j+1) \\ F(i,j+1) \end{array}$$

For each state, we choose one of the three options and call the recursive function again with updated indices:

- If characters at indices i and j match, move indices for both strings $i+1$ and $j+1$.
- Move the index $i+1$ for $string\ 1$.
- Move the index $j+1$ for $string\ 2$.

$$result \rightarrow \begin{array}{c} topDown(str1, str2, i+1, j, memo) \\ topDown(str1, str2, i+1, j+1, memo) \\ topDown(str1, str2, i, j+1, memo) \end{array}$$

Step 3. Choose Optimal Solution
After coming from recursive calls, we choose an optimal solution among these calls and add the value for the current state.

$$F(i,j) \; = \; max\begin{pmatrix} F(i+1,j+1) \; + \; 1 \\ F(i+1,j) \\ F(i,j+1) \end{pmatrix}$$

We get the maximum among all recursive calls and add one if characters at indices i and j match, meaning that we are increasing the longest common subsequence by one.

$$result \rightarrow max\begin{pmatrix} topDown(str1, str2, i+1, j, memo) \\ topDown(str1, str2, i+1, j+1, memo) + 1 \\ topDown(str1, str2, i, j+1, memo) \end{pmatrix}$$

Step 4. Base Case
When we reach a base case, we need to quit a recursive function. The base case can be a base state or case when a sub-problem is no longer valid (in this case, when it returns the worst result).

When the index for at least one of the strings reaches a string length, the function reaches the base case.

```
if (i == text1.length() || j == text2.length()) {
    return 0; // base case
}
```

Step 5. Memoization
We use transforming state variables for memorization.

The index i for $string$ 1 and the index j for $string$ 2 are state variables transforming each state. We can use them for memoization.

```
if (memo[i][j] != -1) {
    return memo[i][j];
}
```

Step 6. Return Result
Return a result of a recursive function.

```
return result
```

Solution

```
int topDown(string& text1, string& text2, int i, int j, vector<vector<int>>& memo) {
    if (i == text1.length() || j == text2.length()) {
        return 0; // base case
    }

    if (memo[i][j] != -1) {
        return memo[i][j];
    }

    int result = INT_MIN;

    if (text1[i] == text2[j]) {
        result = max(result, topDown(text1, text2, i+1, j+1, memo) + 1);
    }

    result = max({result, topDown(text1, text2, i+1, j, memo), topDown(text1, text2, i,
j+1, memo)});

    memo[i][j] = result;

    return result;
}
```

Analysis

Time-Complexity: $O(nm)$

We are calling the function three times and have a total of nm states, where n is the length of $string\ 1$ and m is the length of $string\ 2$.

Space-Complexity: $O(nm)$

We are using nm space to store sub-state solutions.

Bottom-Up

Step 1. Base Case
Find a base case for a given problem.

For the base case, assume that both strings are empty. In that case, the longest common subsequence we can get is 0.

```
int dp[n+1][m+1];
memset(dp, 0, sizeof(dp));
```

Step 2. Move Toward a Target State
Starting from a base case, we move toward a target state.

We need to iterate over both of the strings to compare characters at indices i and j.

```
for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= m; ++j) {
        /* code */
    }
}
```

Step 3. Iterate Over Options
For every step (state), we have several options (ways) to choose to come to the current state. We need to iterate all possible options to find the most optimal solution for the current state.

For each state, there are three options to come to the current state:

- If characters at these indices i and j match, we get the possible best result if we ignore these characters $(i-1, j-1)$.
- If characters at these indices don't match, we get the possible best result if we ignore the current character of $string\ 1$ $(i-1, j)$.
- If characters at these indices don't match, we get the possible best result if we will ignore the current character of $string\ 2$ $(i, j-1)$.

$$dp[i][j] \rightarrow \begin{matrix} dp[i-1][j-1] \\ dp[i-1][j] \\ dp[i][j-1] \end{matrix}$$

Step 4. Choose an Optimal Solution
We find an optimal solution (maximum, minimum, shortest, longest, etc.) without the current state and add a value for the current state.

If characters at indices i and j match, we add one for the current state, and we find the maximum among all options.

```
for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= m; ++j) {
        if (text1[i-1] == text2[j-1]) {
            dp[i][j] = dp[i-1][j-1] + 1;
        }
        dp[i][j] = max({dp[i][j], dp[i-1][j], dp[i][j-1]});
    }
}
```

Step 5. Return Result
After reaching a target state, return an answer for the target state.

We return the answer for the target state.

```
return dp[n][m]
```

Solution

```
int bottomUp(string text1, string text2) {
    int n = (int)text1.length();
    int m = (int)text2.length();

    int dp[n+1][m+1];

    memset(dp, 0, sizeof(dp));

    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            if (text1[i-1] == text2[j-1]) {
                dp[i][j] = dp[i-1][j-1] + 1;
            }
            dp[i][j] = max({dp[i][j], dp[i-1][j], dp[i][j-1]});
        }
    }

    return dp[n][m];
}
```

Analysis

Time-Complexity: $O(nm)$

Inside the nested loop, we are performing constant $O(1)$ operations nm times, where n is the length of $string\ 1$ and m is the length of $string\ 2$.

Space-Complexity: $O(nm)$

We are using nm space to store sub-state solutions.

Best Time to Buy and Sell Stock

Given an array of stock prices on a given day, find the maximum profit you can make by buying and selling stocks. You are allowed to participate in one transaction at a time (you can sell only if you buy before).

Intuition

For every stock in the array, we need to make a decision that will lead to the maximum profit. We need to decide to sell, buy or ignore a stock on the current day.

Top-Down

Step 1. Iterate Over Options
When we solve a problem recursively, for every sub-problem (state), we have different options (directions) to choose to reach our target. We need to iterate over all possible options to find the most optimal solution.

For each day (state), there are three options to choose from:

- Sell a stock if we have one.
- Ignore the current stock.
- Buy the current stock.

Step 2. Reduce Target
Choosing every option, we reduce a target to a particular value, and we call a recursive function again with the new updated target.

For each day (state), we choose one of the available options:

- If we already have bought a stock, we can sell it.
- We can decide not to perform any operation on this day.
- We can buy a stock on this day.

$$result \rightarrow \begin{matrix} sell \\ skip \\ buy \end{matrix}$$

```
int sell = topDown(prices, index+1, false, memo)
int skip = topDown(prices, index+1, canSell, memo);
int buy = topDown(prices, index+1, true, memo)
```

Step 3. Choose Optimal Solution
After coming from recursive calls, we choose an optimal solution among these calls and add the value for the current state.

We have the following available options:

- If we sell a stock, we earn $prices[day]$ on this day.
- If we decide not to perform any operation on this day, the profit will not change.
- If we buy a stock on this day, we pay $prices[day]$.

We choose the best option among the options above.

```
int sell = maxProfit(prices, index+1, sold+1, bought, memo) + prices[index];
int skip = maxProfit(prices, index+1, sold, bought, memo);
int buy = maxProfit(prices, index+1, 0, bought, memo) - prices[index];
int result = max({buy, sell, skip});
```

Step 4. Base Case
When we reach a base case, we need to quit a recursive function. The base case can be a base state or case when a sub-problem is no longer valid (in this case, when it returns the worst result).

We reach the base case when we iterate over all days.

```
if (index >= prices.size()) {
    return 0;
}
```

Step 5. Memoization
We use transforming state variables for memorization.

The $index$ of the day and the possibility to sell stock $canSell$ are state variables transforming each state. We can use them for memoization.

```
if (memo[index][canSell] != -1) {
    return memo[index][canSell];
}
```

Step 6. Return Result
Return a result of a recursive function.

```
return result
```

Solution

```
int topDown(vector<int>& prices, int index, bool canSell, vector<vector<int>>& memo) {
    if (index >= prices.size()) {
        return 0;
    }

    if (memo[index][canSell] != -1) {
        return memo[index][canSell];
    }

    int sell = INT_MIN;
    int buy = INT_MIN;

    if (canSell) {
        // can sell a product
        sell = maxProfit(prices, index+1, false, memo) + prices[index];
    } else {
        // can buy a product
        buy = maxProfit(prices, index+1, true, memo) - prices[index];
    }

    int skip = maxProfit(prices, index+1, canSell, memo);

    int result = max({buy, sell, skip});
    memo[index][canSell] = result;

    return result;
}
```

Analysis

Time-Complexity: $O(n)$

We call the function three times and have a total of n states, where n is the number of days.

Space-Complexity: $O(n)$

We are using n space to store sub-state solutions.

Bottom-Up

Step 1. Base Case
Find a base case for a given problem.

The base cases are two options for the first day:

- Buy on the first day and pay $prices[0]$.
- Sell and earn on the first day. But on the first day, there isn't any stock to sell, so the profit is 0.

```
dp[0][0] = -prices[0]; // 0 - buy
dp[0][1] = 0; // 1 - sell
```

Step 2. Move Toward a Target State
Starting from a base case, we move toward a target state.

We iterate over prices to make the maximum profit.

```
for (int i = 1; i < prices.size(); ++i) {
    /* code */
}
```

Step 3. Iterate Over Options
For every step (state), we have several options (ways) to choose to come to the current state. We need to iterate all possible options to find the most optimal solution for the current state.

For every state, there are three options to come to the current state:

- If you decide to buy new stock, consider the case when you previously sold a stock and update it with a new price.

  `dp[i][0] = dp[i-1][1] - prices[i]`

- If you decide to sell a stock, consider the case when you previously bought the stock and update it with a new price.

  `dp[i][1] = dp[i-1][0] + prices[i]`

- Ignore the current state (use the previous state when you bought or sold a stock).

```
dp[i][0] = dp[i-1][0]
dp[i][1] = dp[i-1][1]
```

Step 4. Choose an Optimal Solution
We find an optimal solution (maximum, minimum, shortest, longest, etc.) without the current state and add a value for the current state.

We find the maximum among all available options.

```
dp[i][0] = max(dp[i-1][0], dp[i-1][1] - prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-1][0] + prices[i])
```

Step 5. Return Result
After reaching a target state, return an answer for the target state.

We return the answer for the target state.

```
return dp[n-1][1]
```

Solution

```
int bottomUp(vector<int>& prices) {
    if (prices.size() == 0) {
        return 0;
    }

    int n = (int)prices.size();

    vector<vector<int>> dp(n, vector<int> (2));

    dp[0][0] = -prices[0];
    dp[0][1] = 0;

    for (int i = 1; i < prices.size(); ++i) {
        dp[i][0] = max(dp[i-1][0], dp[i-1][1] - prices[i]);
        dp[i][1] = max(dp[i-1][1], dp[i-1][0] + prices[i]);
    }

    return dp[n-1][1];
}
```

Analysis

Time-Complexity: $O(n)$

Inside the single loop, we are performing constant $O(1)$ operations n times, where n is the total number of days.

Space-Complexity: $O(n)$

We are using n space to store sub-state solutions.

Nim Game

Given a pile of stones and two players, each player can remove 1 to 5 stones in one move. The player who will remove the last stone will win. Find out whether player 1 can win the game.

$$n - a\ total\ number\ of\ stones, and\ greater\ than\ 0$$
$$k - a\ number\ of\ stones\ can\ be\ removed$$

Intuition

We simulate the game and find out the case when player 2 loses the game.

Top-Down

Step 1. Iterate Over Options
When we solve a problem recursively, for every sub-problem (state), we have different options (directions) to choose to reach our target. We need to iterate over all possible options to find the most optimal solution.

There are options to remove 1 to k stones for every state, where k is the maximum number of stones, 5. We iterate over these options to find out the case that leads to the win.

```
for (int i = 1; i <= k; ++i) {
    /* code */
}
```

Step 2. Reduce Target
Choosing every option, we reduce a target to a particular value, and we call a recursive function again with the new updated target.

We try to remove 1 to k stones for every state and call a recursive function with the new updated target.

```
for (int i = 1; i <= k; ++i) {
    topDown(n - i, k);
}
```

Step 3. Choose Optimal Solution
After coming from recursive calls, we choose an optimal solution among these calls and add the value for the current state.

After coming from recursive calls, check if one of the options leads to losing the game. If the option leads to losing the game, we will leave this option for player 2. Consequently, player 1 will win if we will choose this option.

```
for (int i = 1; i <= k; ++i) {
    if (!topDown(n - i, k)) {
        return true;
    }
}
```

Step 4. Base Case
When we reach a base case, we need to quit a recursive function. The base case can be a base state or case when a sub-problem is no longer valid (in this case, when it returns the worst result).

The base case is the case when the game is over.

```
if (n <= k) {
    return true;
}
```

Step 5. Memoization
We use transforming state variables for memorization.

n is a state variable transforming at each state. We can use it for memoization.

```
if (memo[n] != -1) {
    return memo[n] == 1;
}
```

Step 6. Return Result
Return a result of a recursive function.

Solution

```
// n - a total number of stones
// k - a number of stones can be removed

bool topDown(int n, int k) {
    if (n <= k) {
        return true;
    }

    if (memo[n] != -1) {
        return memo[n] == 1;
    }

    for (int i = 1; i <= k; ++i) {
        if (!topDown(n - i, k)) {
            memo[n] = 1;
            return true;
        }
    }

    memo[n] = 0;

    return false;
}
```

Analysis

Time-Complexity: $O(nk)$

We call the function k times and have a total of n states, where n is the total number of stones and k is the number of stones that can be removed.

Space-Complexity: $O(n)$

We are using n space to store sub-state solutions.

Bottom-Up

Step 1. Base Case
Find a base case for a given problem.

Find the base cases for cases we can predict the result.

```
dp[1] = true;
dp[2] = true;
dp[3] = true;
...
dp[k] = true;
```

Step 2. Move Toward a Target State
Starting from a base case, we move toward a target state.

We find the scenarios for every amount of stone from 1 to n stones.

```
for (int i = k+1; i <= n; ++i) {
    /* code */
}
```

Step 3. Iterate Over Options
For every step (state), we have several options (ways) to choose to come to the current state. We need to iterate all possible options to find the most optimal solution for the current state.

For every state, we try to remove 1 to k stones and consider the case with removed stones.

```
for (int i = k+1; i <= n; ++i) {
    for (int j = 1; j <= k; ++j) {
        dp[i-j];
    }
}
```

Step 4. Choose an Optimal Solution
We find an optimal solution (maximum, minimum, shortest, longest, etc.) without the current state and add a value for the current state.

If the state with removed stones leads to losing the game, then player 1 wins the game. Otherwise, player 1 loses the game.

```
for (int i = k+1; i <= n; ++i) {
    for (int j = 1; j <= k; ++j) {
        if (!dp[i-j]) {
            dp[i] = true;
        }
    }
}
```

Step 5. Return Result
After reaching a target state, return an answer for the target state.

We return the answer for the target state.

`return dp[n]`

Solution

```
// n - a total number of stones
// k - a number of stones can be removed

bool bottomUp(int n, int k) {
    if (n <= k) {
        return true;
    }

    vector<int> dp(n+1, false);

    for (int i = 1; i <= k; ++i) {
        dp[i] = true;
    }

    for (int i = k+1; i <= n; ++i) {
        for (int j = 1; j <= k; ++j) {
            if (!dp[i-j]) {
                dp[i] = true;
            }
        }
    }

    return dp[n];
}
```

Analysis

Time-Complexity: $O(nk)$

Inside the nested loop, we are performing constant $O(1)$ operations nk times, where n is the total number of stones and k is the number of stones that can be removed.

Space-Complexity: $O(n)$

We are using n space to store sub-state solutions.

References

1. Bellman, Richard Ernest. *Eye of the hurricane: an autobiography*. World Scientific. (1984).
2. Bellman, Richard Ernest, *The Theory of Dynamic Programming*. Santa Monica, CA: RAND Corporation, 1954. https://www.rand.org/pubs/papers/P550.html. Also available in print form.