Learning Objectives

Learners will be able to...

- **Differentiate dynamic programming from recursive and iterative solutions**
- **Differentiate between memoization and tabulation**
- **Compare iterative and recursive approach to a problem**

Limits of Recursion

Fibonacci Series

Let's return to a recursive algorithm for the Fibonacci sequence. Copy and paste this code into the IDE to the left.

```
    public static int fib(int n) {
        if (n == 1 || n == 0) {
            return n;
        }
        return fib(n-1) + fib(n-2);
    }
```

First, run the program when asking for the fifth number in the Fibonacci sequence. The program compiles and runs relatively quickly.

Now, let's find the fortieth number in the sequence. In the `main` method, change the value of `n` to `40`. Run the program once more. You should notice that it is a bit slower than before.

```
        int n = 40;
```

Finally, let's call the `fib` method several times, each time calculating the fortieth number in the Fibonacci sequence. The results should be dramatically slower than the ones from above.

```
        System.out.println("Fibonacci number at position " + n +
        " is: " + fib(n));
        System.out.println("Fibonacci number at position " + n +
        " is: " + fib(n));
        System.out.println("Fibonacci number at position " + n +
        " is: " + fib(n));
        System.out.println("Fibonacci number at position " + n +
        " is: " + fib(n));
        System.out.println("Fibonacci number at position " + n +
        " is: " + fib(n));
```

Despite the usefulness of recursive algorithms, they are not always the most performant solution. However, there are things that we can do to increase the performance of recursion.

challenge

Try This Variation

Change `n` to a large number like `100`. You do not need to print out the results of the function call multiple times.

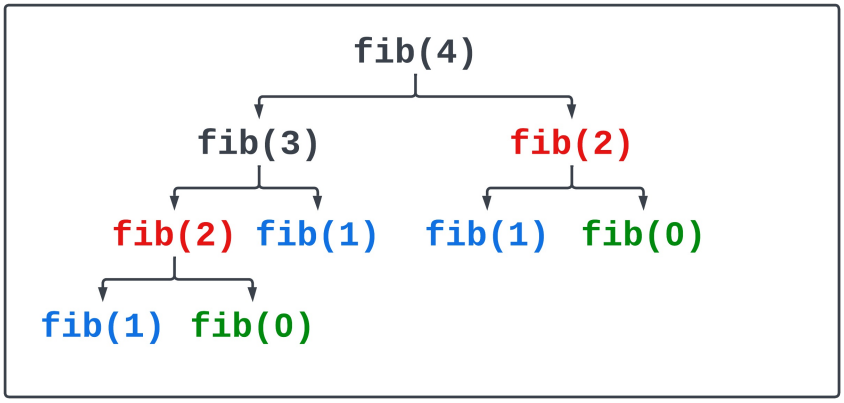
```
        int n = 100;
```

▼ **What is happening?**
We previously discussed that algorithms that make too many recursive calls can fill up the memory on the stack. The timeout message you see is a safety precaution that keeps programs from causing a stack overflow error.

Dynamic Programing

Why is `fib()` so Slow?

The time complexity of the `fib` algorithm from the previous page is approximately $O(2^n)$. Here is a recursive tree for `fib(4)`:



Even though we are only looking for the fourth number in the sequence, our algorithm is making nine recursive calls. This number is so big because several calculations are being repeated. We see `fib(2)` two times, `fib(1)` three times, and `fib(0)` two times. If we did not have to repeat ourselves so many times, our algorithm would not be so slow. That is where dynamic programming comes into play.

Dynamic Programming

Dynamic programming is a technique for solving problems by breaking them down into smaller subproblems, storing the solutions to the subproblems, and then using the stored solutions to solve the original problem. This allows us to solve the original problem more efficiently, since we do not have to repeat any calculations.

Here is an example of how dynamic programming can be used to solve the Fibonacci sequence problem. Break down the problem into subproblems, identifying the base case(s) and the recursive case.

- The first Fibonacci number is 0 (base case).
- The second Fibonacci number is 1 (base case).
- The nth Fibonacci number is the sum of the $(n-1)th$ and $(n-2)th$ Fibonacci numbers (recursive case).

We do not need to perform any calculations for the base case. Those solutions are hard-coded into our algorithm. For every recursive case, we are going to perform the calculation and then store that value. For this example, we are going to use a table to represent storing the calculations. The table might look like this:

Value of n	**Stored Value**
2	1
3	2
4	3
5	5

Once we have the solutions to the first few subproblems, we can solve the original problem by looking up the solution for the desired nth Fibonacci number in the table. For example, the solution to the 5th Fibonacci number is 5, which can be found in the table.

Dynamic programming is a powerful technique that can be used to solve many different kinds of problems. It is often much more efficient than other techniques, such as brute force or backtracking. We are going to use Fibonacci to show dynamic programming using recursion and iteration.

Memoization

Applying Dynamic Programming Principles

To increase the performance of our algorithm, we are going to apply some dynamic programming principles. In particular, we are going to use memoization. **Memorization** is an optimization technique used to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again. This is commonly referred to as a top-down approach.

Let's start by creating the `FibonacciRecursive` class. In it, we need a `main` method and a `fib` method. In addition, we are going to create the global variable `memo` (an array of integers) for the memoization.

```
public class FibonacciRecursive {
    private static int[] memo;

    public static int fib(int n) {

    }

    public static void main(String[] args) {

    }
}
```

Next, we are going to create the variable `n` and set it to 40. Remember, we saw how finding the 40th number in the Fibonacci sequence caused our code to slow down. Set the size of `memo` to one larger than the size of `n`. To help us keep track of which elements have been filled with a Fibonacci number, initialize all of the elements of `memo` to `-1`. Finally, print a message that provides some context to the output of `fib`.

```java
public class FibonacciRecursive {
    private static int[] memo;

    public static int fib(int n) {

    }

    public static void main(String[] args) {
        int n = 40;
        memo = new int[n+1];

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

        System.out.println("Fibonacci number at position " + n +
        " is: " + fib(n));
    }
}
```

Create the `fib` method that takes an integer as a parameter. Before we check our base case, we first are going to see if the calculation was previously performed and stored in the `memo` array. If the element at `memo[n]` is not a `-1`, then it is the result of `fib(n)`. Return this value. Now we can get to the base and recursive cases. If `n` is less than or equal to `1`, then store `n` in `memo[n]`. In the recursive case, set `memo[n]` to the traditional recursive calls for the Fibonacci sequence. Finally, return `memo[n]`.

```java
public class FibonacciRecursive {
    private static int[] memo;

    public static int fib(int n) {
        if (memo[n] != -1) {
            return memo[n];
        }

        if (n <= 1) {
            memo[n] = n;
        } else {
            memo[n] = fib(n-1) + fib(n-2);
        }

        return memo[n];
    }

    public static void main(String[] args) {
        int n = 40;
        memo = new int[n+1];

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

        System.out.println("Fibonacci number at position " + n +
        " is: " + fib(n));
    }
}
```

If we run our code now, it should have no trouble at all with calculating `fib(100)`.

challenge

Try This Variation

Modify algorithm so that it can calculate the 100th number in the Fibonacci sequence. Print it out several times. **Hint**, you need to change the data type for the `fib()` method to be a `long` since the 100th number is too big for the `int` data type.

▼ **Solution**

If `fib()` now returns a `long`, then `memo` also needs to be of type `long`. This includes when you create the variable and assign it a length.

```
public class FibonacciRecursive {
    private static long[] memo;

    public static long fib(int n) {
        if (memo[n] != -1) {
            return memo[n];
        }

        if (n <= 1) {
            memo[n] = n;
        } else {
            memo[n] = fib(n-1) + fib(n-2);
        }

        return memo[n];
    }

    public static void main(String[] args) {
        int n = 100;
        memo = new long[n+1];

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

        System.out.println("Fibonacci number at position " +
        n + " is: " + fib(n));
        System.out.println("Fibonacci number at position " +
        n + " is: " + fib(n));
        System.out.println("Fibonacci number at position " +
        n + " is: " + fib(n));
        System.out.println("Fibonacci number at position " +
        n + " is: " + fib(n));
        System.out.println("Fibonacci number at position " +
        n + " is: " + fib(n));
    }
}
```

Analysis

The use of dynamic programming principles means that the time complexity of `fib()` is now $O(n)$. The space complexity becomes a bit more complicated as we now introduce greater auxiliary space due to memoization. The space complexity for the memoization table is $O(n)$, and the space complexity for the recursive call stack is also $O(n)$.

Although the recursive approach seems more complicated, it's a good illustration of the power of memoization and how a naive (it is called "naive" because calculations are forgotten after each recursive call) recursive solution can be optimized using dynamic programming principles.

Iterative Dynamic Programming

Iterative Fibonacci Sequence

Recursion is not the only way to calculate a number from the Fibonacci sequence, and an iterative approach also stands to benefit from implementing dynamic programming. Instead of starting from a high number and breaking it down into smaller sub-problems (like recursion), we begin from the smallest problems and build-up. Create the method `fib()` that returns an integer and also takes an integer as a parameter. Just like the recursive approach, we know that `fib(0)` or `fib(1)` do not require any calculations. Return `0` or `1` in these respective cases.

```
    public static int fib(int n) {
        if (n <= 1) {
            return n;
        }
    }
```

If `n` is greater than `1`, then we need to create an array of integers of size `n + 1` to store our calculations. **Note**, since there is no recursion, we do not need to make this array a global variable. Instead, it can reside in the method because `fib()` will only be called once. Set `fib(0)` to `0` and `fib(1)` to `1`. This way we are ready to start calculating values for when `n` is greater than or equal to `2`.

```
    public static int fib(int n) {
        if (n <= 1) {
            return n;
        }

        int[] memo = new int[n + 1];
        memo[0] = 0;
        memo[1] = 1;
    }
```

Finally, iterate from `2` (because the first two elements in the array already have values) up to and including `n`. For each value `i`, calculate the Fibonacci number using the two previous elements in the array. After the loop has finished, return the element at index `n`.

```java
public static int fib(int n) {
    if (n <= 1) {
        return n;
    }

    int[] memo = new int[n + 1];
    memo[0] = 0;
    memo[1] = 1;

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

    return memo[n];
}
```

Just as before, we are able to calculate `fib(40)` without much of a delay.

Compared to the recursive counterpart, the iterative `fib()` method is a more efficient algorithm. The time complexity is still $O(n)$, but the space complexity is reduced. You still have a memoization table with a space complexity of $O(n)$, but there is no recursive call stack. The iterative `fib()` is only called one time.

The above approach efficiently computes the Fibonacci of `n` in linear time. This is much faster than the naive recursive method, especially for large values of `n`.

Tabulation

Dynamic Programming with a 2D Array

Another way to implement dynamic programming is through tabulation. That is, we use a 2D array to store data in a table, where the elements are arranged into rows and columns. Each cell in the table contains solutions to all of the sub-problems that make up a larger problem. This is commonly referred to as a bottom-up approach.

Two-dimensional arrays (**2D arrays**) are essentially arrays of arrays. They are extremely useful for representing data that naturally fits in a table, such as a chess board or a grid. In dynamic programming, 2D arrays can be used to store values that correspond to solutions of sub-problems in a grid-like fashion. Let's look at an example to illustrate this idea.

Longest Common Subsequence (LCS)

Suppose we are trying to find the length of the longest common subsequence between two strings `A` and `B`. A subsequence is a sequence of characters that appear in the same order in the string, but not necessarily consecutively. For example, look at the following strings:

```
"BACBDAB"
"BDCAB"
```

The longest common subsequence for these strings would be `"BCAB"`. Calculating the length of the longest common subsequence is a common problem in computer science as it lends itself to dynamic programming. Using a naive recursive approach would result in many redundant calculations. To more efficiently solve this problem, we are going to use an iterative approach with tabulation.

With dynamic programming, we can store the solutions to sub-problems in a 2D array `dp`, where `dp[i][j]` will contain the length of LCS of prefixes `A[0..i-1]` and `B[0..j-1]`.

Start by creating the `longestCommonSubsequence` method which takes two strings and returns an integer. Calculate the length of each string, and create the 2D array `dp` where each array is one element longer than their respective strings.

```
public class LCS {
    public static int longestCommonSubsequence(String A, String
        B) {
        // Get the length of each input string.
        int m = A.length();
        int n = B.length();

        // Initialize a 2D array to store the LCS lengths.
        // dp[i][j] will contain the length of LCS of prefixes
        A[0..i-1] and B[0..j-1].
        int[][] dp = new int[m+1][n+1];

    }
```

Iterate over the two arrays. If any of the cells are in the first row (`i == 0`) or first column (`j == 0`) then set the value of the element to `0`.

```
public class LCS {
    public static int longestCommonSubsequence(String A, String
        B) {
        // Get the length of each input string.
        int m = A.length();
        int n = B.length();

        // Initialize a 2D array to store the LCS lengths.
        // dp[i][j] will contain the length of LCS of prefixes
        A[0..i-1] and B[0..j-1].
        int[][] dp = new int[m+1][n+1];

        // Loop through each cell in the 2D array.
        for(int i = 0; i <= m; i++) {
            for(int j = 0; j <= n; j++) {
                // Base case: If either of the strings is empty,
        LCS length is 0.
                if(i == 0 || j == 0) {
                    dp[i][j] = 0;
                }

    }
```

It will be helpful to visualize what this table looks like and how it relates to the strings. Here is the table with the first row and column filled with zeros. Each character of the first string represents a row, while the characters in the second string represent the columns.

		B	D	C	A	B
	0	0	0	0	0	0
B	0					
A	0					
C	0					
B	0					
D	0					
A	0					
B	0					

If the cell in the table is not in the first row or column, then determine if the two characters are the same. If so, set the value of the current cell to the value of the cell up one row and one column to the left plus 1.

```
public class LCS {
    public static int longestCommonSubsequence(String A, String
        B) {
        // Get the length of each input string.
        int m = A.length();
        int n = B.length();

        // Initialize a 2D array to store the LCS lengths.
        // dp[i][j] will contain the length of LCS of prefixes
        A[0..i-1] and B[0..j-1].
        int[][] dp = new int[m+1][n+1];

        // Loop through each cell in the 2D array.
        for(int i = 0; i <= m; i++) {
            for(int j = 0; j <= n; j++) {
                // Base case: If either of the strings is empty,
        LCS length is 0.
                if(i == 0 || j == 0) {
                    dp[i][j] = 0;
                }
                // If the corresponding characters in A and B
        are equal,
                // then we can extend the LCS for A[0..i-2] and
        B[0..j-2] by 1.
                else if(A.charAt(i-1) == B.charAt(j-1)) {
                    dp[i][j] = dp[i-1][j-1] + 1;
                }

    }
```

In our example, the two strings both start with the character “B”. Our current cell (the blue cell) represents the comparison of characters “B” and “B”. Since they are the same, take the value of the red cell (up one row and one column to the left) and increment by `1`. That means the value of the blue cell is `1` since the value of the red cell was `0`.

		B	D	C	A	B
	0	0	0	0	0	0
B	0	1				
A	0					
C	0					
B	0					
D	0					
A	0					
B	0					

If the two characters that are being compared are different, then the value of the current cell becomes whatever value is the largest between the cell directly to the left and the cell directly above.

```java
public class LCS {
    public static int longestCommonSubsequence(String A, String
        B) {
        // Get the length of each input string.
        int m = A.length();
        int n = B.length();

        // Initialize a 2D array to store the LCS lengths.
        // dp[i][j] will contain the length of LCS of prefixes
        A[0..i-1] and B[0..j-1].
        int[][] dp = new int[m+1][n+1];

        // Loop through each cell in the 2D array.
        for(int i = 0; i <= m; i++) {
            for(int j = 0; j <= n; j++) {
                // Base case: If either of the strings is empty,
        LCS length is 0.
                if(i == 0 || j == 0) {
                    dp[i][j] = 0;
                }
                // If the corresponding characters in A and B
        are equal,
                // then we can extend the LCS for A[0..i-2] and
        B[0..j-2] by 1.
                else if(A.charAt(i-1) == B.charAt(j-1)) {
                    dp[i][j] = dp[i-1][j-1] + 1;
                }
                // If the corresponding characters in A and B
        are different,
                // take the maximum LCS length between A[0..i-1]
        & B[0..j-2] or A[0..i-2] & B[0..j-1].
                else {
                    dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
                }
            }
        }

    }
```

The next iteration has the current cell (blue) comparing the characters “B” and “D”. Since they are not the same, look to the cell to the left and the cell directly above (red). Take the larger of these two values and assign it to the current cell.

		B	D	C	A	B
	0	0	0	0	0	0
B	0	1	1			
A	0					
C	0					
B	0					
D	0					
A	0					
B	0					

As you continue to fill out the table in this manner, you will see the following as the final result:

		B	D	C	A	B
	0	0	0	0	0	0
B	0	1	1	1	1	1
A	0	1	1	1	2	2
C	0	1	1	2	2	2
B	0	1	1	2	2	3
D	0	1	2	2	2	3
A	0	1	2	2	3	3
B	0	1	2	2	3	4

Continue iterating through the array and, when the loops have finished, return the element in the bottom-right corner of the table. This cell represents the length of the longest common subsequence between the two strings.

```java
public class LCS {
    public static int longestCommonSubsequence(String A, String
        B) {
        // Get the length of each input string.
        int m = A.length();
        int n = B.length();

        // Initialize a 2D array to store the LCS lengths.
        // dp[i][j] will contain the length of LCS of prefixes
        A[0..i-1] and B[0..j-1].
        int[][] dp = new int[m+1][n+1];

        // Loop through each cell in the 2D array.
        for(int i = 0; i <= m; i++) {
            for(int j = 0; j <= n; j++) {
                // Base case: If either of the strings is empty,
        LCS length is 0.
                if(i == 0 || j == 0) {
                    dp[i][j] = 0;
                }
                // If the corresponding characters in A and B
        are equal,
                // then we can extend the LCS for A[0..i-2] and
        B[0..j-2] by 1.
                else if(A.charAt(i-1) == B.charAt(j-1)) {
                    dp[i][j] = dp[i-1][j-1] + 1;
                }
                // If the corresponding characters in A and B
        are different,
                // take the maximum LCS length between A[0..i-1]
        & B[0..j-2] or A[0..i-2] & B[0..j-1].
                else {
                    dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
                }
            }
        }

        // The value at dp[m][n] contains the length of LCS for
        A and B.
        return dp[m][n];
    }
```

In this code, we initialized a 2D array `dp` with dimensions `(m+1) x (n+1)`, where `m` and `n` are the lengths of strings `A` and `B`, respectively. We then filled up the array row by row, using the solutions to smaller sub-problems to solve larger sub-problems. Finally, `dp[m][n]` contains the length of the LCS of `A` and `B`.

By using a 2D array, we have eliminated the need for redundant calculations, and this program will run in $O(m * n)$ time, which is much more efficient than the naive recursive approach.

Formative Assessment 1

Formative Assessment 2