Fibonacci Solutions

The program aims to resolve the value of a target position within the Fibonacci sequence. The solution uses two approaches to resolve this value: recursive and dynamic programming.

# Recursive

The recursive algorithm decomposes a target value to produce calls of 0 or 1 which return a static value. Each recursive call resolves the value from the previous call in the stack. At the bottom, the function resolves to 0 or 1. Each subsequent call takes the result of the previous (n – 1) and sums that with the result of the value of iteration before that (n – 2). This creates a branching tree of function calls where the base nodes are 0 or 1.

# Dynamic Programming

The DP algorithm requires the first two positions in the sequence be hard-coded. These values are known as 0 and 1. The function then iterates from the second position referencing index n-1 and index n-2 within the array the function is building. Using an array instead of a recursive function to resolve the values allows for quicker access to the data and computation subsequently reducing the time complexity of the algorithm versus the recursive approach.

# Implementation

using System.Diagnostics;

internal class Program

{

// Recursively produce fib sequence value at position (target)

static int RecursiveFibSeq(int target) {

// Targets of 0 and 1 are 0 or 1 so just return target if the value

// is 0 or 1

if (target <= 1) return target;

// Recursively call this function reducing the value target to eventually

// return a 0 or a 1. The sum will bubble up

return RecursiveFibSeq(target - 1) + RecursiveFibSeq(target - 2);

}

// Use dynamic programming to produce sequence value at position (target)

static int DPFibSeq(int target) {

// create an array of size (target + 1) which will

// store the calculated sequence values including

// the 0th position

int[] sequence = new int[target + 1];

// prepare array with 0th and 1st values to begin

// calculating from 2nd value onward

sequence[0] = 0;

sequence[1] = 1;

// iterate to (target) to find that position's value

// starting at the 2nd position since first two already

// declared

for (int i = 2; i <= target; i++) {

// calculate position i using previous two values

sequence[i] = sequence[i - 1] + sequence[i - 2];

}

return sequence[target];

}

static void Main(string[] args)

{

Stopwatch stopWatch = new Stopwatch();

// Set the desired position within a fibonacci sequence

int position = <N>;

// Capture length of runtime for recursive execution

stopWatch.Start();

int fibRecursive = RecursiveFibSeq(position);

stopWatch.Stop();

// Format the resulting time to a readable timestamp

TimeSpan ts = stopWatch.Elapsed;

string recursiveElapsedTime = String.Format("{0:00}:{1:00}:{2:00}.{3:00}",

ts.Hours, ts.Minutes, ts.Seconds,

ts.Milliseconds / 10);

// Reset the stop watch otherwise it resumes at previous time

stopWatch.Reset();

// Capture the length of runtime for dynamic programming execution

stopWatch.Start();

int fibDp = DPFibSeq(position);

stopWatch.Stop();

// Format the resulting time to a readable timestamp

ts = stopWatch.Elapsed;

string dpElapsedTime = String.Format("{0:00}:{1:00}:{2:00}.{3:00}",

ts.Hours, ts.Minutes, ts.Seconds,

ts.Milliseconds / 10);

// Output simple report with the position, sequence result, and time for each method

Console.WriteLine($"Fibonacci sequence to position {position}");

Console.WriteLine($"- Recusive result: {fibRecursive}\t\t\tTook {recursiveElapsedTime}");

Console.WriteLine($"- Dynamic Programming result: {fibDp}\tTook {dpElapsedTime}");

}

}

# Time Complexity

The recursive solution results in an exponential increase in runtime, or in other words it has a time complexity of O(2n). Each call results in two additional calls resulting in 2n method calls where n—or the *target* variable—is greater than 1.

The dynamic programming solution results in linear time complexity O(n). The method uses a data structure to build the sequence. Accessing the array completes in constant time. However, the method must iterate n times where n is the value of the *target* variable.

The following chart illustrates the difference in execution.