Volume

1

ITERATIVE VS. RECURSIVE METHOD TIMING

JAVA PROGRAMMING User Manual

Java Fibonacci Programming Manual

Table of Contents

Chapter

1

Fibonacci Numbering Sequence

"The Fibonacci Sequence turns out to be the key to understanding how nature designs… and is… a part of the same ubiquitous music of the spheres that builds harmony into atoms, molecules, crystals, shells, suns, and galaxies and makes the Universe sing."
- Guy Murchie

In this manual we are going to show you how we can output the Fibonacci Sequence of numbers by using 2 different approaches; Iterative and Recursive. We will also be timing the sequence to find out which method is faster to use. The Fibonacci numbering sequence is a special kind of sequence that starts with a 0 and 1, and every number after those two is the sum of the two numbers preceding it.

For Example: 0, 1, 1, 2, 3, 5, 8, 13, 21, … and so on… This pattern is said to repeat itself in nature.

We will only use 1 class and will be putting our output under the public static main void(Sting[] args). We are trying to see which approach is faster for Java programming. **n** will represent our input on the x-axis and the y-axis will represent the time it took in nano seconds to get the output. We will start at input 20 and will end at input 40. All together there will be a total of 21 inputs. This will give us a good idea at which method is quicker to use for Java programming to use for Fibonacci Sequencing.

To find the timing between the two numbering sequences we will be using this formula below to find the nano seconds:

long startTime = System.nanoTime();
long endTime = System.nanoTime();
(endTime – startTime) = time it took to find the answer (put in System.out)

Chapter 2

Recursive Java Process

The Recursive method code we will be using is shown below:

```
static int fibRecursive(int n)
if (n == 0) return 0;
if (n == 1) return 1;
return fibRecursive(n-1) + fibRecursive(n-2);
```

Since we will be putting a timer into this method for the input **n**. We will see how long it takes to compile the answer in nano seconds. I will not be converting the nano seconds into seconds, since this will not give us an exact answer on how long it is taking this method to get the desired output for our input **n** on the x-axis.

The time complexity for this approach is 0(2^N) which is considered an exponential time complexity, where n is the index of the nth Fibonacci number in the sequence.

We will use a simple incremental for loop for our input **n**. This for loop will also be used for the Iterative method. The for loop is shown in the example below, where n = 40 as the input for the increment to end at.

```
int n = 40;
for (int I = 20; I <= n; ++i); {
}
```

This for loop will start our input off at 20 and will work its way until it reaches 40 as the last input. Each output will give us the time it took the formula to achieve its outcome on the y-axis. Because of the time and space complexities of this program, it must call the function 2 times for each value.

Chapter 3

Iterative Java Process

The Iterative method we will be using is shown below:

```
Static int fibIterative(int n) {
int v1 = 0, v2 = 1, v3 = 0;
for (int i = 2; I <= n; i++); {
v3 = v1 + v2;
v1 = v2;
v2 = v3;
}
return v3
}
```

The time complexity for this approach is 0(N) which is considered a linear space and time complexity. This means that the space/memory being used remains the same, and or constant through the duration of the process.

We will also be putting a timer on this process during its output. This will allow us to see how long it took to complete its iterations. The idea behind this process is to find out which process is faster by breaking the time down to nano seconds.

Chapter 4

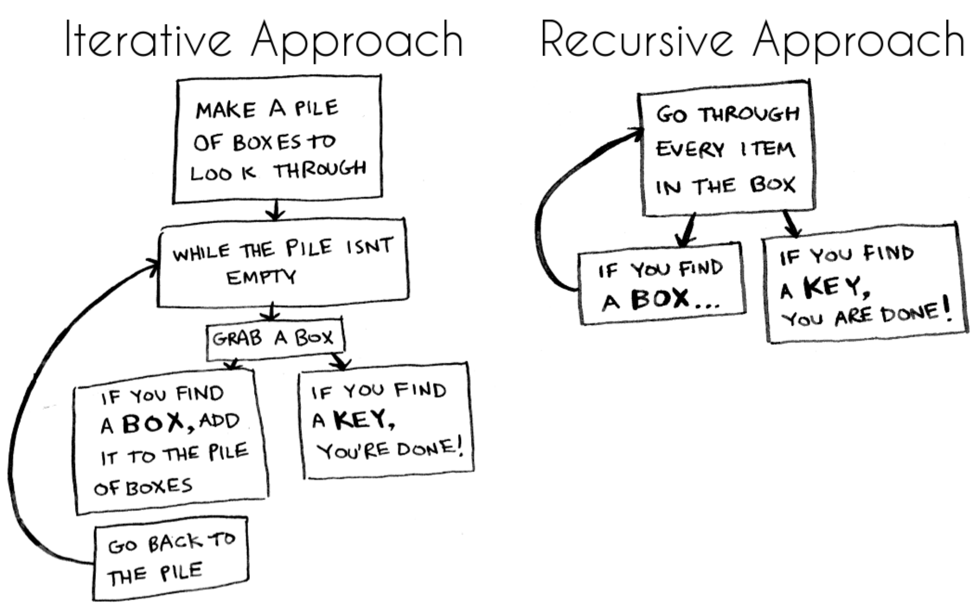
Results

To get the results of the input we will be using this code shown below:

fibRecursive(i);
System.out.print(i + "\t" + (System.nanoTime – startTime);

fibIterative(i);
System.out.print(System.nanoTime() – startTime);

For the sake of understanding the output, I left out the added white space from code shown above. The timer will calculate the time it took to process the integer **i** in fibRecursive(i) and fibIterative(i). To understand how the two methods work separately in achieving the results, there is a picture displayed below showing the difference:



The image above shows an easy to read and understand illustration as to why the recursive program method is more efficient in speed for processing the Fibonacci sequence. It does this by showing how a box is broken down by the 2 different methods. During my testing, the output was also showed that recursive was faster. I ran three separate tests, with the inputs of 20 incrementing to 40. Those results are shown below:

x-axis	Recursive	Iterative
20	2409750	2670542
21	311083	363792
22	361750	406000
23	291833	306375
24	743833	818375
25	887708	923000
26	1215833	1231666
27	3087583	3159750
28	1421916	1445458
29	2236459	2255959
30	3632958	3652667
31	6268542	6328542
32	9962542	10022209
33	16108833	16170750
34	26914292	26993000
35	42906125	43006792
36	70849709	71027000
37	122172917	122262834
38	198529417	198603625
39	310790375	310862166
40	494638792	494734959

x-axis	Recursive	Iterative
20	1052166	1159125
21	1349417	1416208
22	196250	210500
23	288292	299875
24	687375	771125
25	820375	863000
26	1327042	1348667
27	2595708	2683042
28	4444042	4596000
29	2262458	2332583
30	3639916	3665458
31	6196917	6268000
32	9575083	9647125
33	15053917	15120375
34	26362083	26438292
35	44106500	44248875
36	67866042	67932667
37	107260083	107338208
38	173074792	173151792
39	285154625	285227833
40	461641125	461730792

x-axis	Recursive	Iterative
20	2128542	3340959
21	123292	140834
22	206917	240083
23	299167	314292
24	593000	627792
25	764542	787667
26	1288708	1439541
27	2036125	2077834
28	5880209	5971459
29	2137625	2163125
30	3348959	3368125
31	6002709	6067667
32	10976041	11046291
33	14799416	14887833
34	25791958	25909125
35	41600083	41711083
36	70608709	70673917
37	112451625	112534875
38	176247000	176324750
39	283428000	283497542
40	457413417	457503584

Recursive has shown that it is a faster method for processing the Fibonacci sequence of numbers. There are some known cases when the Iterative method can potentially be faster when using certain memorization compartmentalizing techniques, but for this example recursive is the winner.

In the next chapter, I will publish my source code for the user to try out and look over. I was using Eclipse as my IDE and did not have any issues with compiling the code.

Chapter

5

Source Code

Shown below is my entire source code for this example. I included all the white space measures I used in getting the results. I also added note lines for you to understand the code a little bit better.

```
package cen3024c;

public class FibonacciTiming {

        static int fibRecursive(int n) { //initializing input for int n
                        if (n == 0) return 0;
                        if (n == 1) return 1;
                        return fibRecursive(n-1) + fibRecursive(n-2);
                }

        static int fibIterative(int n) { //initializing input for int n
                int v1 = 0, v2 = 1, v3 = 0;
                for (int i = 2; i <= n; i++); {
                v3 = v1 + v2;
                v1 = v2;
                v2 = v3;
                }
                return v3;
                }

public static void main(String[] args) {

        System.out.print("x-axis\tRecursive\tIterative\n");
        System.out.println("------\t---------\t---------");
        int n = 40; // for loop input as n
        for (int i = 20; i <= n; ++i) { // incremental for loop to start at 20, all the way to 40
                long startTime = System.nanoTime(); //setting long startTime to nano seconds

        fibRecursive(i); /*passing the input i from the incremental for loop through the fib
recursive method*/
                System.out.print(i + "\t" + (System.nanoTime() - startTime) +"  ");

        fibIterative(i); /*passing the input i from the incremental for loop through the fib
recursive method*/
                System.out.print("\t" + (System.nanoTime() - startTime) + "\n");
        }
        }
        }
```