Fibonacci: Recursion vs. Iteration

1. Complete the Fibonacci methods below. Write TWO answers for this problem: one using recursion and one using iteration.

/\*\*\*\*\*\*\*\*\*\*

pre-condition: *n* > 0

post-condition: returns the *n*th Fibonacci #: 1, 1, 2, 3, 5, 8, 13,..

\*\*\*\*\*\*\*\*\*\*/

|  |  |
| --- | --- |
| public static long fibRecur(int n)  { | public static long fibIter(int n)  { |
| if (n <= 2)  return 1;  return (fibRecur(n - 1) + fibRecur(n - 2)); | long n1 = 1;  long n2 = 1;  long val = 1;  if (n == 1 || n == 2) {  return 1;  }  for (int i = 3; i <= n; i++) {  val = n2;  n2 = n1 + n2;  n1 = val;  }  return n2; |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| } | } |

|  |  |
| --- | --- |
| 2. Trace the recursive calls for fibRecur(5).Diagram, schematic  Description automatically generated | 3. Which do you think will be quicker, Fibonacci by recursion or Fibonacci by iteration? Why?  Fibonacci by iteration will be much quicker. This is because iteration is keeping accumulated values saved and works forwards until the number is reached. On the other hand, recursion keeps making calls to itself, and this takes a lot more processing power than just accessing variables. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

4. Complete the shell and record both run-times for the given Fibonacci numbers. Graph both curves on the graph paper on the back. What kinds of curves are they? Explain why the results make sense.

Both are linear curves. The recursive curve has a much larger slope, and the computation time increases linearly as the input number increases. The iterative function has a curve closer to zero, and any slope seems insignificant when graphed with the recursive function. The computation time for the linear function remains mostly constant.

5. Complete the hailstone methods below. Use recursion. Each is recursive, but counts the steps in different ways. For the input 12, output is: 12-6-3-10-5-16-8-4-2-1 takes 10 steps

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

pre-condition: n > 0

post-condition: prints the hailstone sequence that starts with n.

Counts and returns the number of steps from n to 1.  
If n is even then the next number is n / 2.

If n is odd then the next number is 3 \* n + 1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

|  |  |
| --- | --- |
| //recursive, prints the sequence, counts the // steps with a variable  public static int hailstone(int n, int count)  { | //recursive, print the sequence, //counts the steps without a variable  public static int hailstone(int n)  { |

|  |  |
| --- | --- |
| if (n != 1)  if (n % 2 == 0) {  count += 1;  return hailstone(n / 2, count);  } else {  count += 1;  return hailstone(3 \* n + 1, count);  }  return count; | if (n != 1) {  if (n % 2 == 0) {  return hailstone(n / 2) + 1;  } else {  return hailstone(3 \* n + 1) + 1;  }  }  return 1; |