Fibonacci sequence starts with 0 and 1 and the next number is the sum of the two preceding numbers. I have provided an example below starting with 0 and 1.

0 1 1 2 3 5 8 13

Recursion is the technique of making a function call itself. To implement the Fibonacci function in a recursive fashion a static int is made (FibonacciRe) and we would pass in and integer, n.

```
public static int fibonacciRe(int n) {


    if(n<=1)
    {
    return 1;
    }
    else {
    return fibonacciRe(n-1)+fibonacciRe(n-2);
    }
}
```

If we passed in 5 for n we return 4 + 3 which would give us the next number in the sequence. This is recursive because we are making a call on to itself, n.

Iteration is a technique used to sequence through code until over and over until a condition is met. To implement Fibonacci function in an iterative fashion fibonaciIt is created and we would pass in integer, n.

```
public static int fibonacciIt(int n)

        if(n<=1)
        {
            return 1;
        }

    int r=0,p=1,pp=1,i;

    for(i=2;i<=n;i++){
        r = p + pp;
        pp =p;
        p =r;
    }
        return r;
}
```

If we passed 5 for n the for loop would continue until the condition was not true which would be 4 iterations. The loop starts with 0 and 1 and would return 2 for the first iteration and would continue the loop until i < == n is not true. If for loop was printed with 5 passed in for n it would display 2 3 5 8. I have displayed the values of r, p and pp through each iteration 4 , n(5).

r	2	3	5	8
p	2	3	5	8
pp	1	2	3	5

System.*nanoTime*(); will provide the time in nano. Long start_time, long_endtime are created to mark the start and end of the recursive and iterative Fibonacci methods. The variables differenceIt and differenceRe are created to store the run time for the functions which be explained more later.

```
System.out.print("n\tRuntimeIterative\tRuntimeRecursive\n");
System.out.print("==\t==============\t==============\n");

long start_time,end_time;
int differenceIt,differenceRe;


for(i=10;i<=n;i++) // 10 to 40
{
start_time = System.nanoTime(); //start
f2 = fibonacciRe(i); //calling recursive method
end_time = System.nanoTime(); //end
differenceRe = (int)((end_time - start_time) / 1e6); //end - start

start_time = System.nanoTime(); //start
f1 = fibonacciIt(i); //calling iterative method
end_time = System.nanoTime(); //end
differenceIt = (int)((end_time - start_time) / 1e6); //end - start


System.out.print(i+"\t\t"+differenceIt+"\t\t\t"+differenceRe +"\t\n");
```

Once methods are called we subtract end time and start time of the method and divide 1e6 to receive nano seconds and they are displayed line by line in until the loop ends. The loop starts at 10 and we pass in 40. Below are the results.

```
<terminated> fibo [Java Application] C:\Users\Rigel Kent\.p2\pool
n       RuntimeIterative        RuntimeRecursive
==      ==============  ==============
10              0                       0
11              0                       0
12              0                       0
13              0                       0
14              0                       0
15              0                       0
16              0                       0
17              0                       0
18              0                       0
19              0                       0
20              0                       0
21              0                       0
22              0                       0
23              0                       0
24              0                       0
25              0                       0
26              0                       0
27              0                       0
28              0                       1
29              0                       1
30              0                       3
31              0                       5
32              0                       8
33              0                       13
34              0                       21
35              0                       35
36              0                       56
37              0                       91
38              0                       148
39              0                       240
40              0                       390
```

Iterative is the fastest because of the use of variables in the method and it does need to call back on itself. In recursion method the stack space slows down the process.