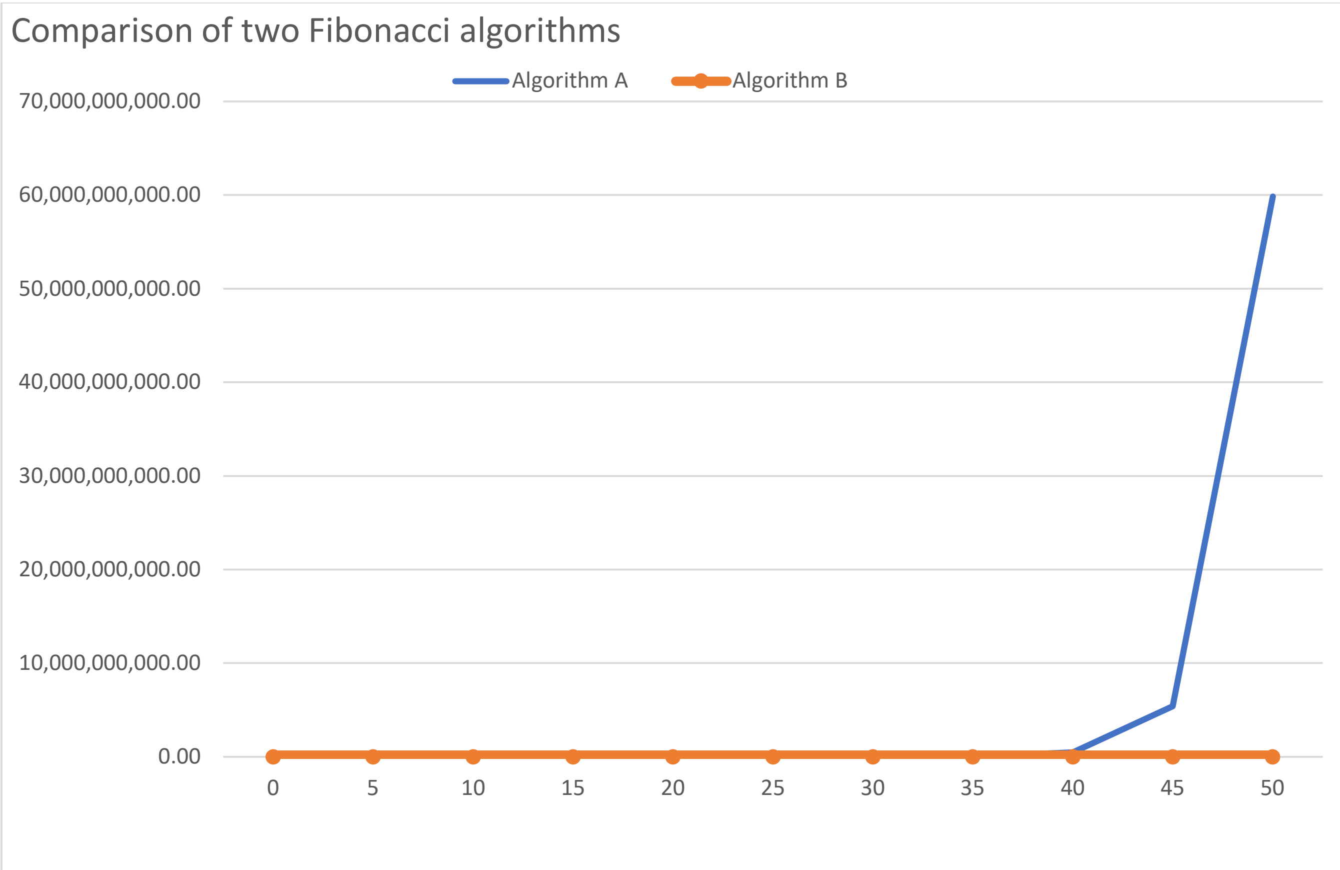
YAVUZ SELİM SARIBAŞ 22003920 CS101 HOMEWORK-2



An instance of the result of the test code;

Plot: "test number + result+ *n* values + calculation time"

1   0 // for n=0 ,F_0 is calculated in 1099 nanoseconds.

2   5 // for n=5 ,F_5 is calculated in 801 nanoseconds.

3   55 // for n=10 ,F_10 is calculated in 15000 nanoseconds.

4   610 // for n=15 ,F_15 is calculated in 129100 nanoseconds.

5   6765 // for n=20 ,F_20 is calculated in 140300 nanoseconds.

6   75025 // for n=25 ,F_25 is calculated in 623701 nanoseconds.

7   832040 // for n=30 ,F_30 is calculated in 3958399 nanoseconds.

8   9227465 // for n=35 ,F_35 is calculated in 44132200 nanoseconds.

9   102334155 // for n=40 ,F_40 is calculated in 489409600 nanoseconds.

10   1134903170 // for n=45 ,F_45 is calculated in 5393003601 nanoseconds.

11   12586269025 // for n=50 ,F_50 is calculated in 59872278300 nanoseconds.

----

1   0 // for n=0 ,F_0 is calculated in 8399 nanoseconds.

2   5 // for n=5 ,F_5 is calculated in 6901 nanoseconds.

3   55 // for n=10 ,F_10 is calculated in 999 nanoseconds.

4   610 // for n=15 ,F_15 is calculated in 5001 nanoseconds.

5   6765 // for n=20 ,F_20 is calculated in 4399 nanoseconds.

6   75025 // for n=25 ,F_25 is calculated in 3700 nanoseconds.

7   832040 // for n=30 ,F_30 is calculated in 3201 nanoseconds.

8   9227465 // for n=35 ,F_35 is calculated in 3600 nanoseconds.

9   102334155 // for n=40 ,F_40 is calculated in 5100 nanoseconds.

10   1134903170 // for n=45 ,F_45 is calculated in 3099 nanoseconds.

11 12586269025 // for n=50 ,F_50 is calculated in 2000 nanoseconds.

As clearly seen in the results, recursion is way slower than while loop calculator since the recursive method call is overdone, and therefore, call stack usage is enormously increased. Java should store method parameters, local variables for each recursive method call to maintain the state of the method before that call. “Java isn’t performing tail call optimization either, so don’t count on it.” Hence, ignoring that Java doesn’t optimize Tail Recursion, loops such as while, do or for, must reasonably be faster compared to the recursion in this example.

The other linked problem is what happens if we increase *n* from 50 to a greater number, say 100. It has a probability of getting an error, which is stack overflow. Because the larger number we choose, the more recursion we do. Therefore, computers require more memory allocation for much more new stack frames

for example on calcFibA() method, lets say, n = 5;

in this sutation, function calls first calcFibA(5) and it returns calcFibA(4)+ calcFibA(3),

and then , java need to call the methods one by one, therefore , it calculates calcFibA(4), which is

calcFibA(3)+ calcFibA(2).

So we have calcFibA(3)+ calcFibA(2)+ calcFibA(3).

And then calcFibA(3) should be calculated , which is calcFibA(2)+ calcFibA(1).

Then the last calcFibA(3) will calculated , which is again calcFibA(2)+ calcFibA(1).

As a result , we have ;

calcFibA(5) = calcFibA(2)+ calcFibA(1) + calcFibA(2)+ calcFibA(2)+ calcFibA(1).

Lastly, calcFibA(2) = calcFibA(1)+ calcFibA(1).

Therefore we have 5*(calcFibA(1)) = 5

To conclude, every recursion need a new method call stack that is accumulated in memory, and it cause a work overload.