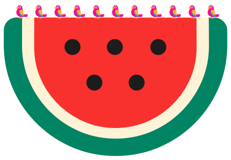
Week11a

- the Fibonacci sequence
- recursion
- dynamic programming



WARMUP

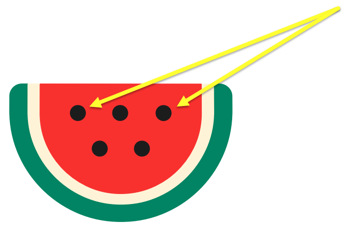
What does this print?

```
PRINT(0.1 + 0.2);
PRINT(SQRT(2.0) * SQRT(2.0));
```

WARMUP

What is F_7?

$F_0 = 0$
$F_1 = 1$
$F_k = F_{k-1} + F_{k-2}$



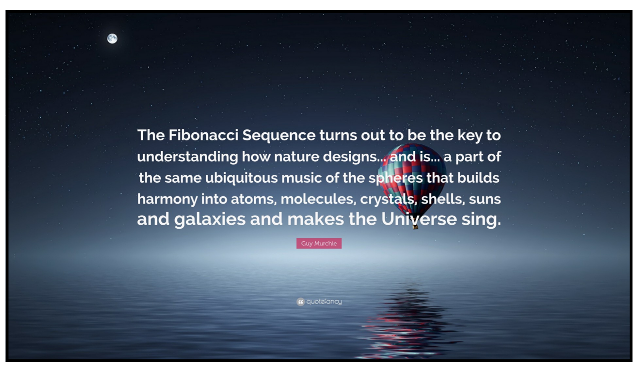
what are these?

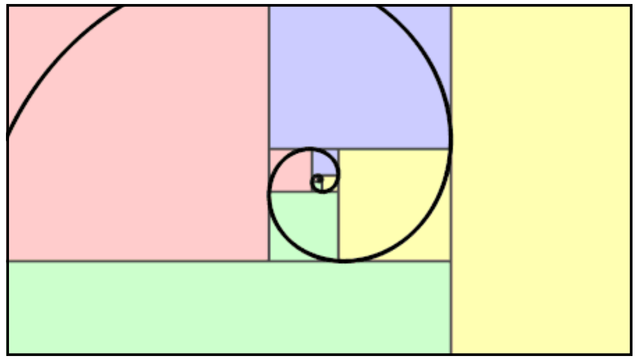
---

the SEEDS OF LEARNING®

---

the Fibonacci sequence





```
// F_0                   =  0
// F_1                   =  1
// F_2 = F_1 + F_0 = 1 + 0 =  1
// F_3 = F_2 + F_1 = 1 + 1 =  2
// F_4 = F_3 + F_2 = 2 + 1 =  3
// F_5 = F_4 + F_3 = 3 + 2 =  5
// F_6 = F_5 + F_4 = 5 + 3 =  8
// F_7 = F_6 + F_5 = 8 + 5 = 13
// ...
```

recursion

review: recursion basics

recursion

- a **recursive function** is a function that calls itself
  - each call must make progress towards a **base case** (when the function finally returns without calling itself)
    - ✨ when in doubt, try something like zero for your base case

```
class Main {
    static int digitSum(int n) {
        if (n == 0) {
            return 0;
        }
        return digitSum(n / 10) + (n % 10);
    }

    public static void main(String[] arguments) {
        PRINT(digitSum(256)); // 13
    }
}
```

```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

```
return 0;
return digitSum(0) + 2;
return digitSum(2) + 5;
return digitSum(25) + 6;
int a = digitSum(256);
```

```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

```
return 0;
return digitSum(0) + 2;
return digitSum(2) + 5;
return digitSum(25) + 6;
int a = digitSum(256);
```

```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

```
return 0 + 2;
return digitSum(2) + 5;
return digitSum(25) + 6;
int a = digitSum(256);
```

```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

```
return 2;
return digitSum(2) + 5;
return digitSum(25) + 6;
int a = digitSum(256);
```

```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

```
return 2;
return digitSum(2) + 5;
return digitSum(25) + 6;
int a = digitSum(256);
```

```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

```
return 2 + 5;
return digitSum(25) + 6;
int a = digitSum(256);
```

```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

```
return 7;
return digitSum(25) + 6;
int a = digitSum(256);
```

```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

```
return 7;
return digitSum(25) + 6;
int a = digitSum(256);
```

```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

```
return 7 + 6;
int a = digitSum(256);
```

```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

```
return 13;
int a = digitSum(256);
```

```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

```
return 13;
int a = digitSum(256);
```

```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

```
int a = 13;
```

fin.

⚠️ recursion hazard 1

repeated computation

🐌 **example:** slow very slow fibonnaci

(couldn't we just use a for loop...?)

```
// F_0 = 0
// F_1 = 1
// F_2 = F_1 + F_0 = 1 + 0 = 1
// F_3 = F_2 + F_1 = 1 + 1 = 2
// F_4 = F_3 + F_2 = 2 + 1 = 3
// F_5 = F_4 + F_3 = 3 + 2 = 5
// ...
static long fib(long k) {
    if (k == 0) { return 0; }
    if (k == 1) { return 1; }
    return fib(k - 1) + fib(k - 2);
}
```

```
fib(4)
```

```
(fib(3) + fib(2))
```

```
((fib(2) + fib(1)) + (fib(1) + fib(0)))
```

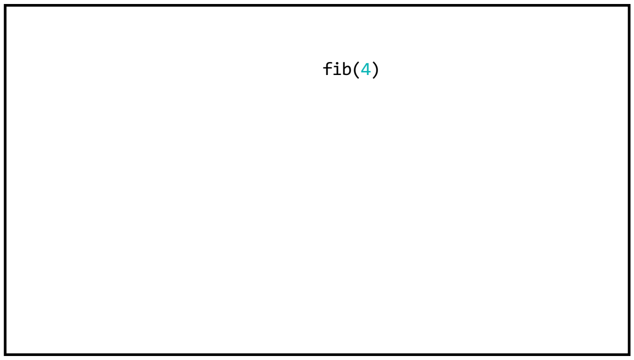
```
(((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0)))
```

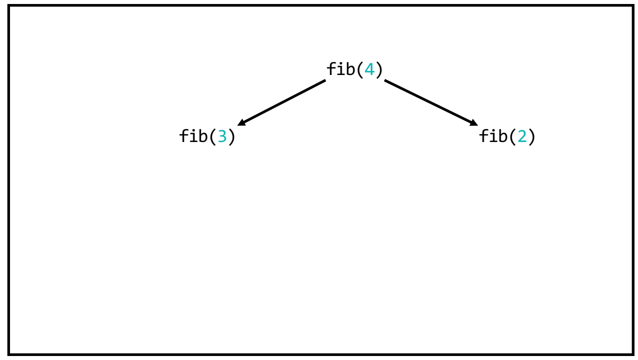
```
(((1 + 0) + 1) + (1 + 0))
```

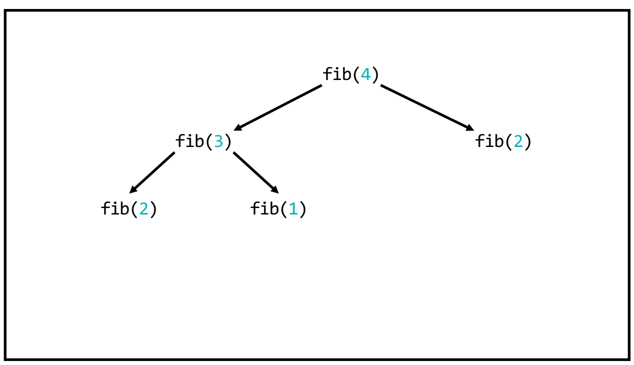
```
((1 + 1) + 1)
```

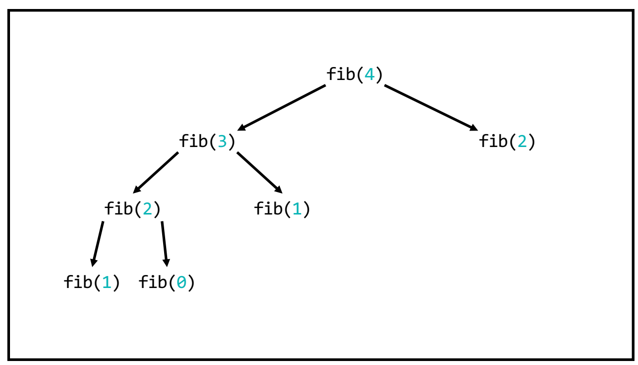
```
(2 + 1)
```

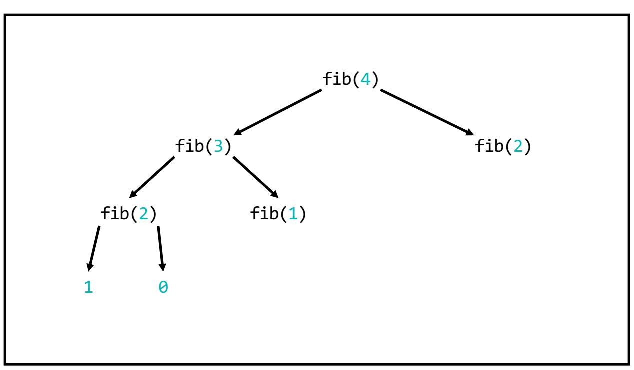
```
3
```

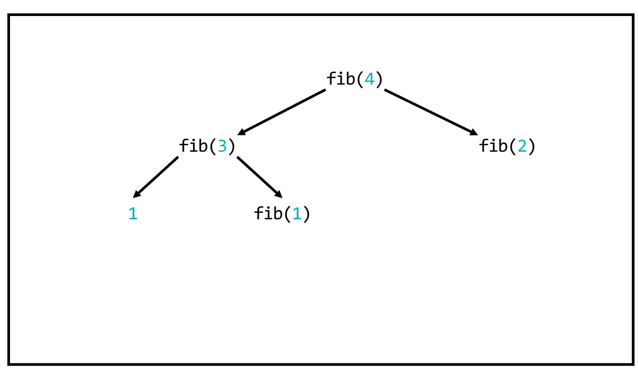

fib(4)

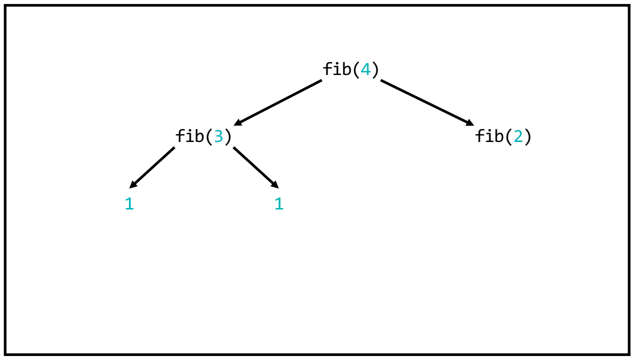

fib(4)
fib(3)
fib(2)

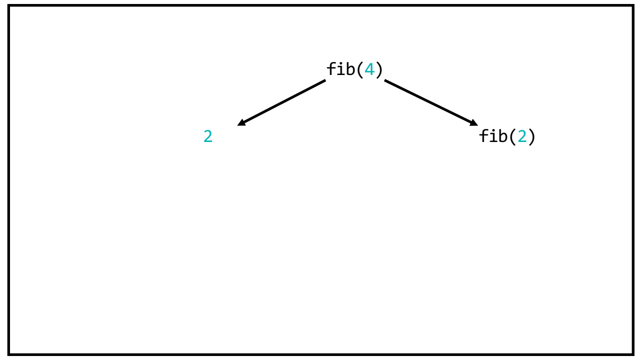

fib(4)
fib(3)
fib(2)
fib(2)
fib(1)

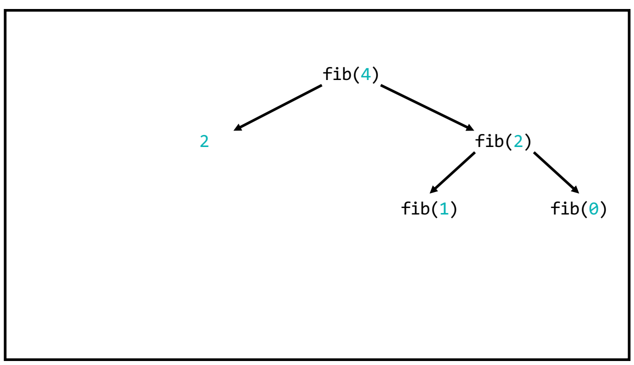

fib(4)
fib(3)
fib(2)
fib(2)
fib(1)
fib(1)
fib(0)

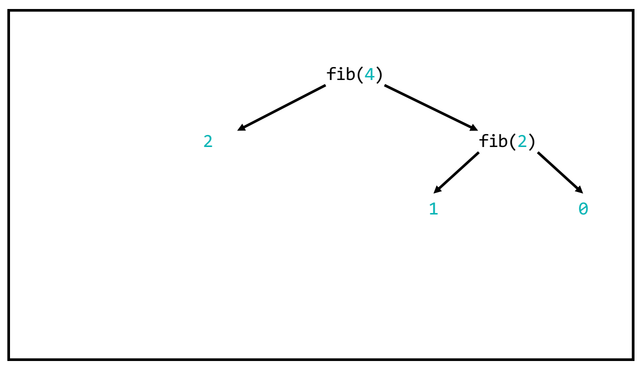

fib(4)
fib(3)
fib(2)
fib(2)
fib(1)
1
0

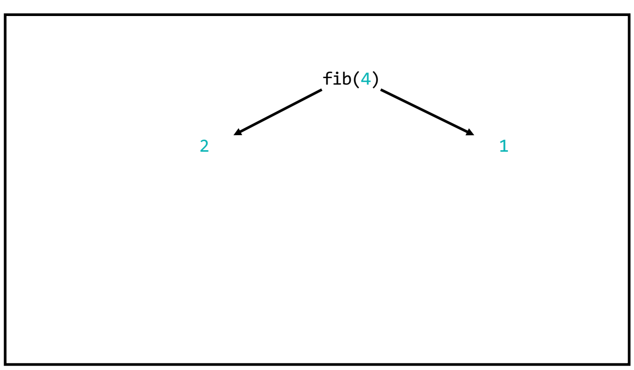

fib(4)
fib(3)
fib(2)
1
fib(1)

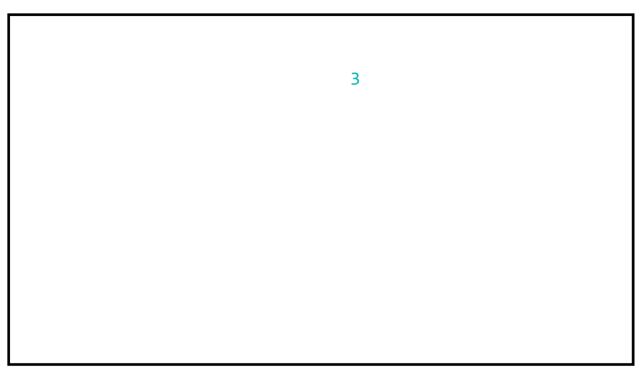

fib(4)
fib(3)
fib(2)
1
1

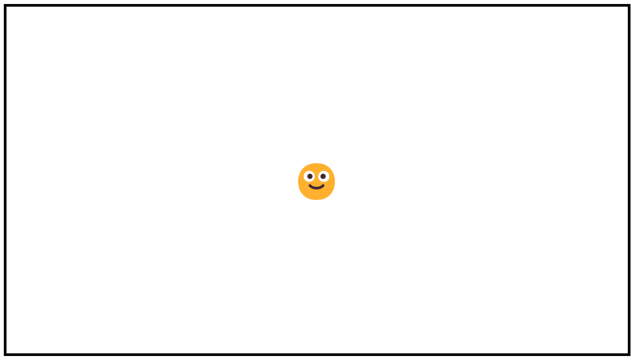

fib(4)
2
fib(2)

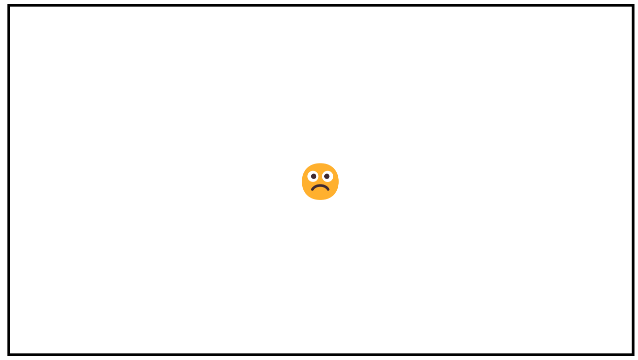

fib(4)
2
fib(2)
fib(1)
fib(0)

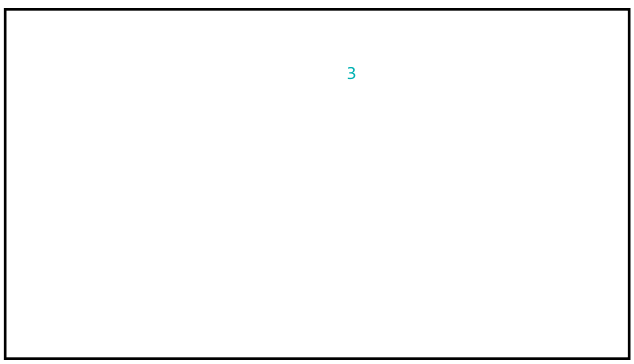

fib(4)
2
fib(2)
1
0

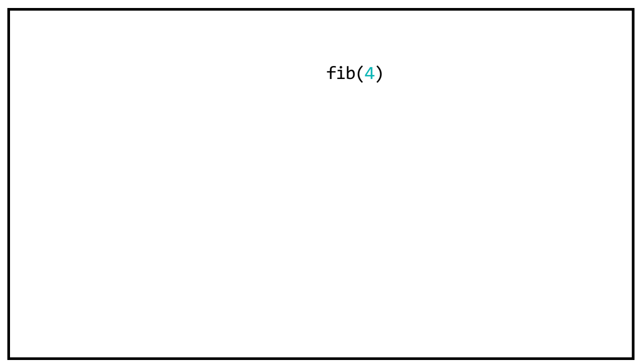

fib(4)
2
1

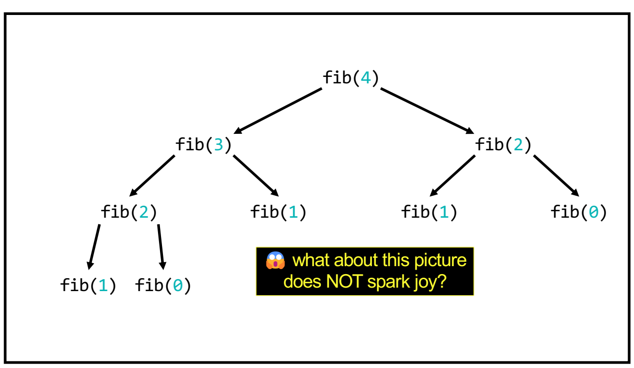

3

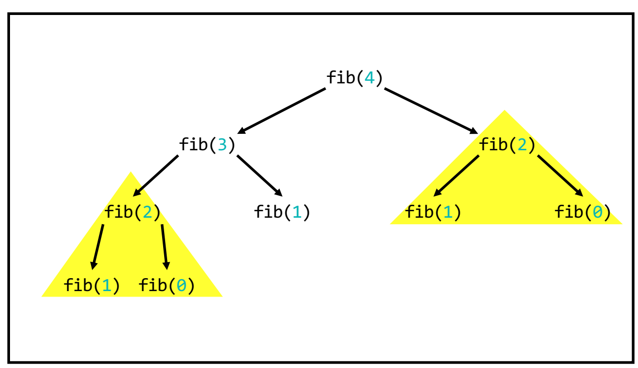





3


fib(4)


fib(4)
fib(3)
fib(2)
fib(2)
fib(1)
fib(1)
fib(0)
fib(1)
fib(0)
😱 what about this picture
does NOT spark joy?

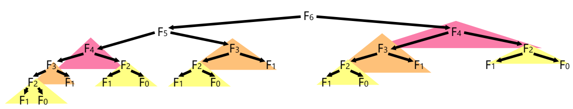

fib(4)
fib(3)
fib(2)
fib(2)
fib(1)
fib(1)
fib(0)
fib(1)
fib(0)

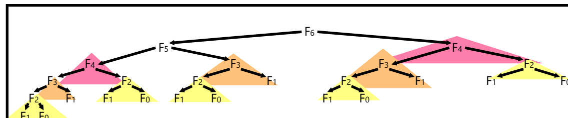
we computed `fib(2)` from scratch two seperate times

😱 we are **repeating computation**!

(what a waste 😞)

and for bigger n...
there is (much) more repetition





🧠 who's that
big O runtime?

exponential 🙀

`[fib(5), fib(36), fib(77) demo]`

⚠️ recursion hazard 2

stack overflow

☠️ 🐌 dangerous slow $\sum_{i=1}^{n} i = 1 + \cdots + n$

```java
// 1 + ... + n
static int sum(int n) {
    if (n == 0) return 0;
    return n + sum(n - 1);
}
```

`sum(100000)`

`sum(99999)`

↑

`sum(100000)`

`sum(0)`

↑

⋮

↑

`sum(99999)`

↑

`sum(100000)`

😱 what about this picture does NOT spark joy?

---

the height of the callstack is O(n)

😱 big n → **stack overflow**!

😞

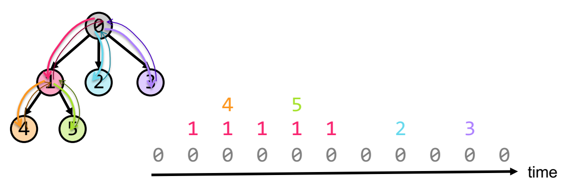
[sum(100000) demo]

our depth-first binary (search) tree traversals were recursive...

...should we be worried about them overflowing the callstack?

no. (assuming your tree is ~balanced)

callstack height for depth-first **balanced** binary tree traversal is log(n)



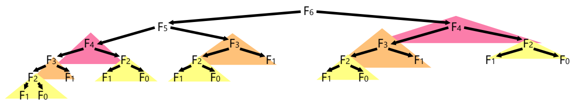
*additionally, some languages/compilers** have "tail-call optimization," which would prevent a stack overflow for `sum(100000)`

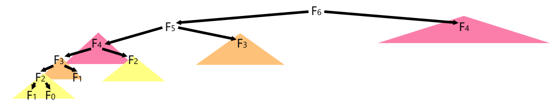
**Java is not one of these languages (as our demo showed)

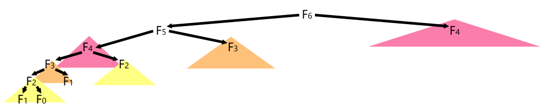
dynamic programming

dynamic programming is when you use the result of previous computation

(this is a squishy definition)







🧠 who's that big O runtime?

linear 🙂 👍

memoization

memoization means storing the results of previous functions calls, so we don't have to repeat work when the function is called again

memoized fibonacci

```java
static HashMap<Long, Long> table = new HashMap<>();

static long memoizedFib(long k) { // NOTE: long is an integer type
                                  //       that can store larger numbers than int
    if (k == 0) { return 0; }
    if (k == 1) { return 1; }

    long Fkm1;
    if (table.containsKey(k - 1)) {
        Fkm1 = table.get(k - 1);
    } else {
        Fkm1 = memoizedFib(k - 1);
        table.put(k - 1, Fkm1);
    }

    long Fkm2;
    if (table.containsKey(k - 2)) {
        Fkm2 = table.get(k - 2);
    } else {
        Fkm2 = memoizedFib(k - 2);
        table.put(k - 2, Fkm2);
    }

    return Fkm1 + Fkm2;
}
```

what's the runtime of each call to memoizedFib(...)?

```java
public static void main(...) {
    int n = ...;
    long a = memoizedFib(n);
    long b = memoizedFib(n - 1);
    long c = memoizedFib(n + 1);
}
```

```java
int n = ...;
long a = memoizedFib(n);      // O(n)
long b = memoizedFib(n - 1);  // O(1)
long c = memoizedFib(n + 1);  // O(1)
```

closed form fibonacci

Computation by rounding [edit]

Since

$$\frac{|\psi|^n}{\sqrt{5}} < \frac{1}{2}$$

for all $n \geq 0$, the number F_n is the closest integer to

$$\frac{\varphi^n}{\sqrt{5}}.$$

Therefore it can be found by rounding, or in terms of the floor function:

$$F_n = \left\lfloor \frac{\varphi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor,\ n \geq 0.$$

Or the nearest integer function:

$$F_n = \left[\frac{\varphi^n}{\sqrt{5}} \right],\ n \geq 0.$$

Similarly, if we already know that the number $F > 1$ is a Fibonacci number, we can determine its index within the sequence by

$$n(F) = \left\lfloor \log_\varphi \left(F \cdot \sqrt{5} + \frac{1}{2} \right) \right\rfloor$$

approximate closed-form fibonnaci

```java
// NOTE: Because of floating point error, this does not work for big n.
//       (On my computer, returns wrong result for n > 70.)
static long closedFormFib(long n) {
    final double goldenRatio = (1.0 + Math.sqrt(5.0)) / 2.0;
    return Math.round(Math.pow(goldenRatio, n) / Math.sqrt(5.0));
}
```

exponentiation by squaring

note: there is actually a log(n) algorithm using matrices and "exponentiation by squaring"

this algorithm does NOT have floating point problems (all numbers are integers)

$$
\begin{array}{cccc}
N=1 & N=2 & N=4 & N=8 \\
\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} & \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix} & \begin{pmatrix} 5 & 3 \\ 3 & 2 \end{pmatrix} & \begin{pmatrix} 34 & 21 \\ 21 & 13 \end{pmatrix} \\
1 & 2 & 4 & 8
\end{array}
$$

Times matrix is multiplied with itself