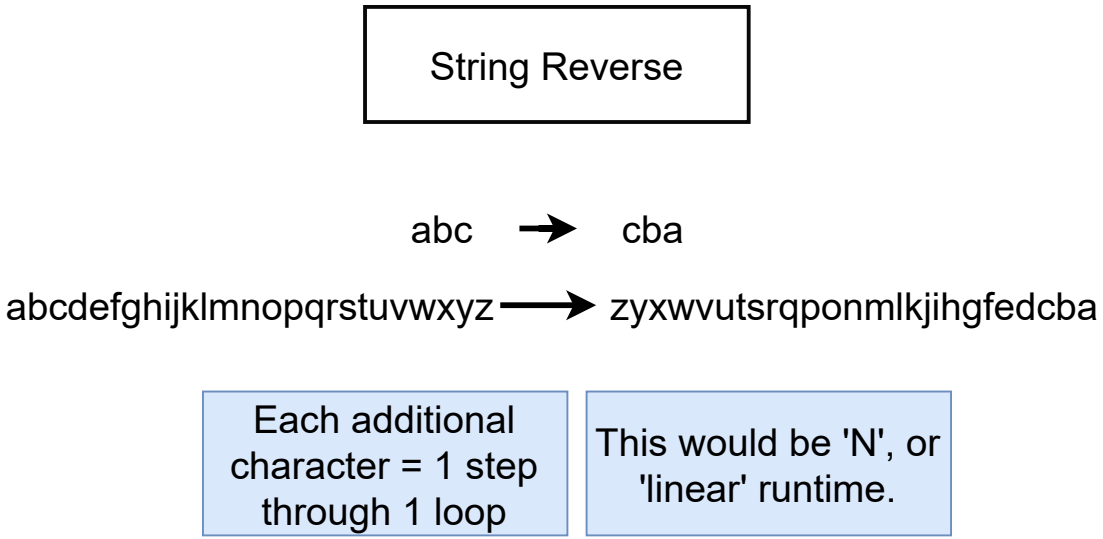
Runtime Complexity

Describes the performance
of an algorithm

How much more processing power/time is required
to run your algorithm if we double the inputs?


String Reverse
abc
cba
abcdefghijklmnopqrstuvwxyz
zyxwvutsrqponmlkjihgfedcba
Each additional character = 1 step through 1 loop
This would be 'N', or 'linear' runtime.

Steps Algorithm

steps = 2

#	-
#	#

Had to do
4 things

steps = 3

#	-	-
#	#	-
#	#	#

Had to do
9 things

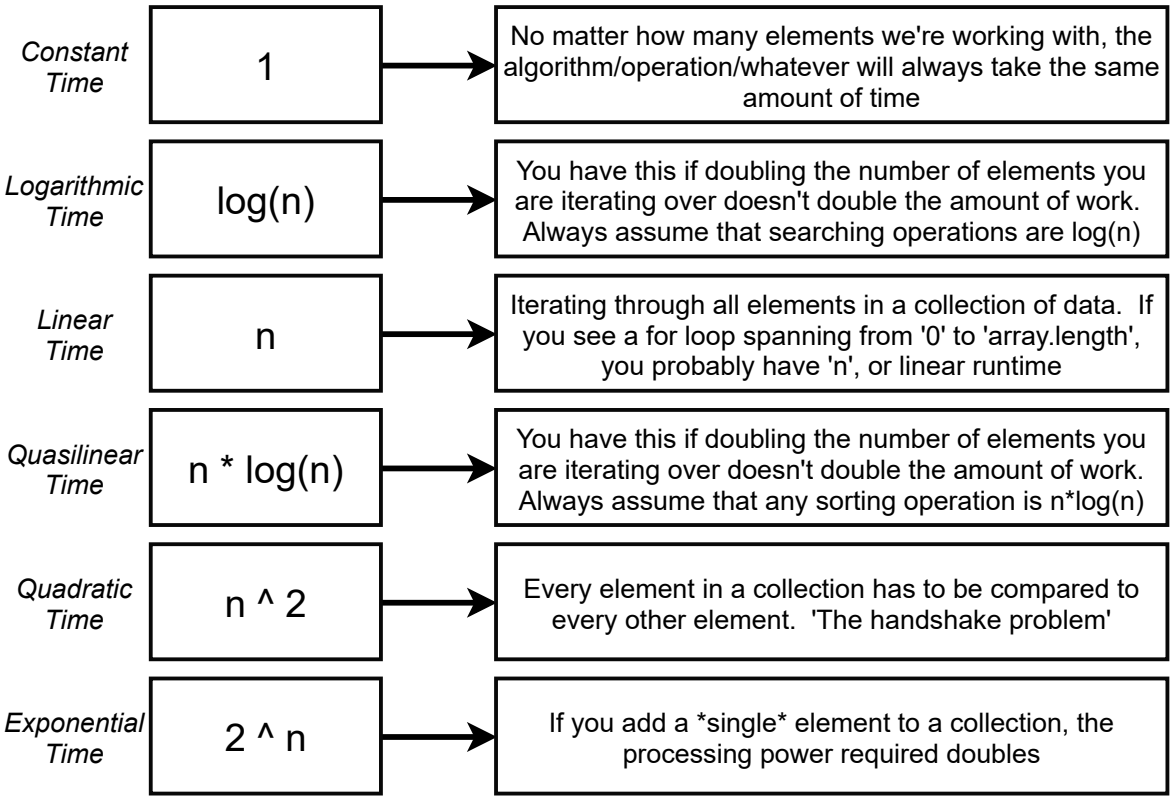
steps = 4

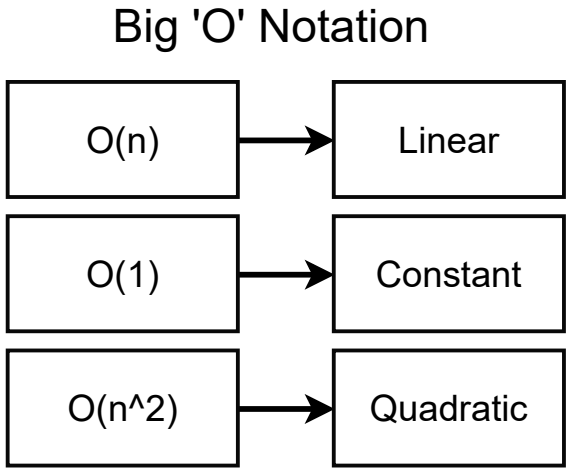
#	-	-	-
#	#	-	-
#	#	#	-
#	#	#	#

Had to do
16 things

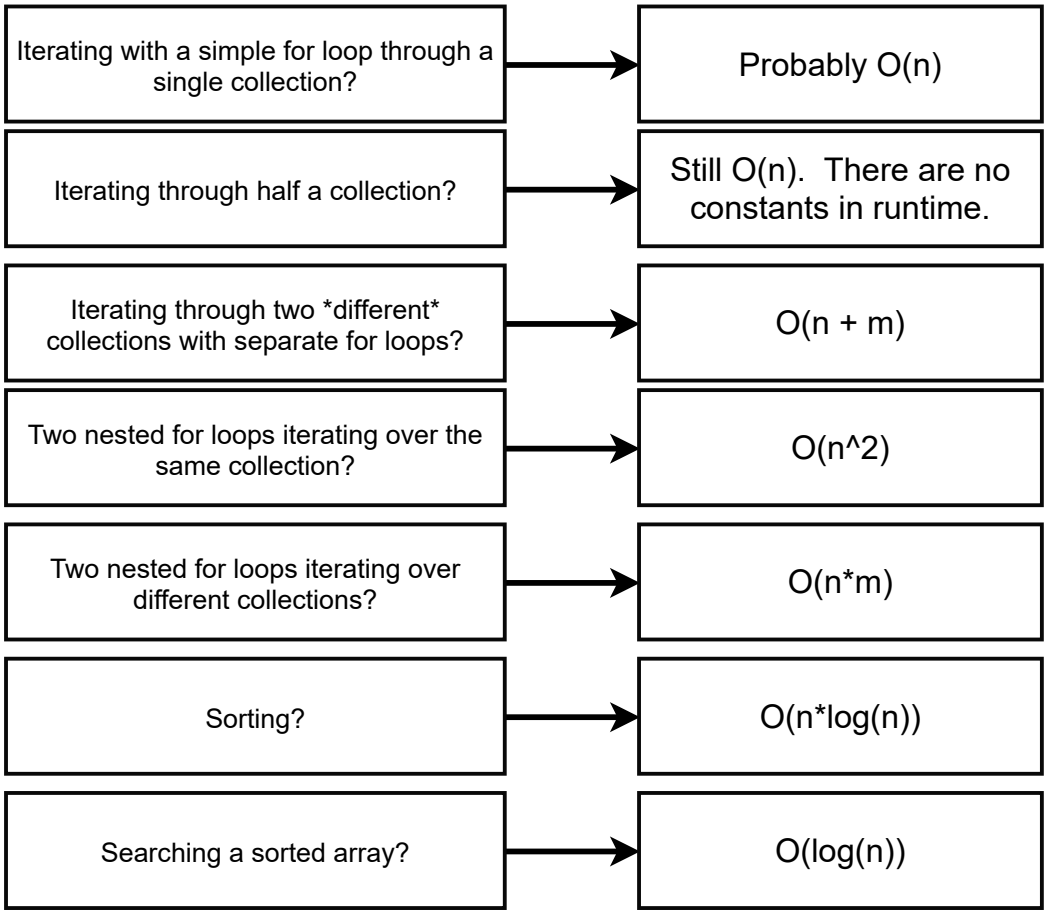
As 'n' increased by one, we had to do way, way more stuff, or (n*n) things total

This would be N^2, or quadratic runtime


Constant Time
1
No matter how many elements we're working with, the algorithm/operation/whatever will always take the same amount of time
Logarithmic Time
log(n)
You have this if doubling the number of elements you are iterating over doesn't double the amount of work. Always assume that searching operations are log(n)
Linear Time
n
Iterating through all elements in a collection of data. If you see a for loop spanning from '0' to 'array.length', you probably have 'n', or linear runtime
Quasilinear Time
n * log(n)
You have this if doubling the number of elements you are iterating over doesn't double the amount of work. Always assume that any sorting operation is n*log(n)
Quadratic Time
n ^ 2
Every element in a collection has to be compared to every other element. 'The handshake problem'
Exponential Time
2 ^ n
If you add a *single* element to a collection, the processing power required doubles


Big 'O' Notation
O(n)
Linear
O(1)
Constant
O(n^2)
Quadratic

Identifying Runtime Complexity

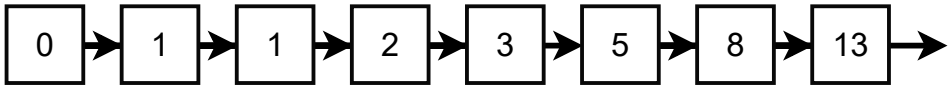


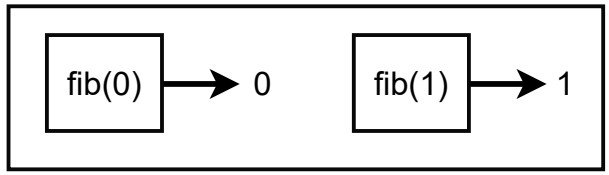
Last note...

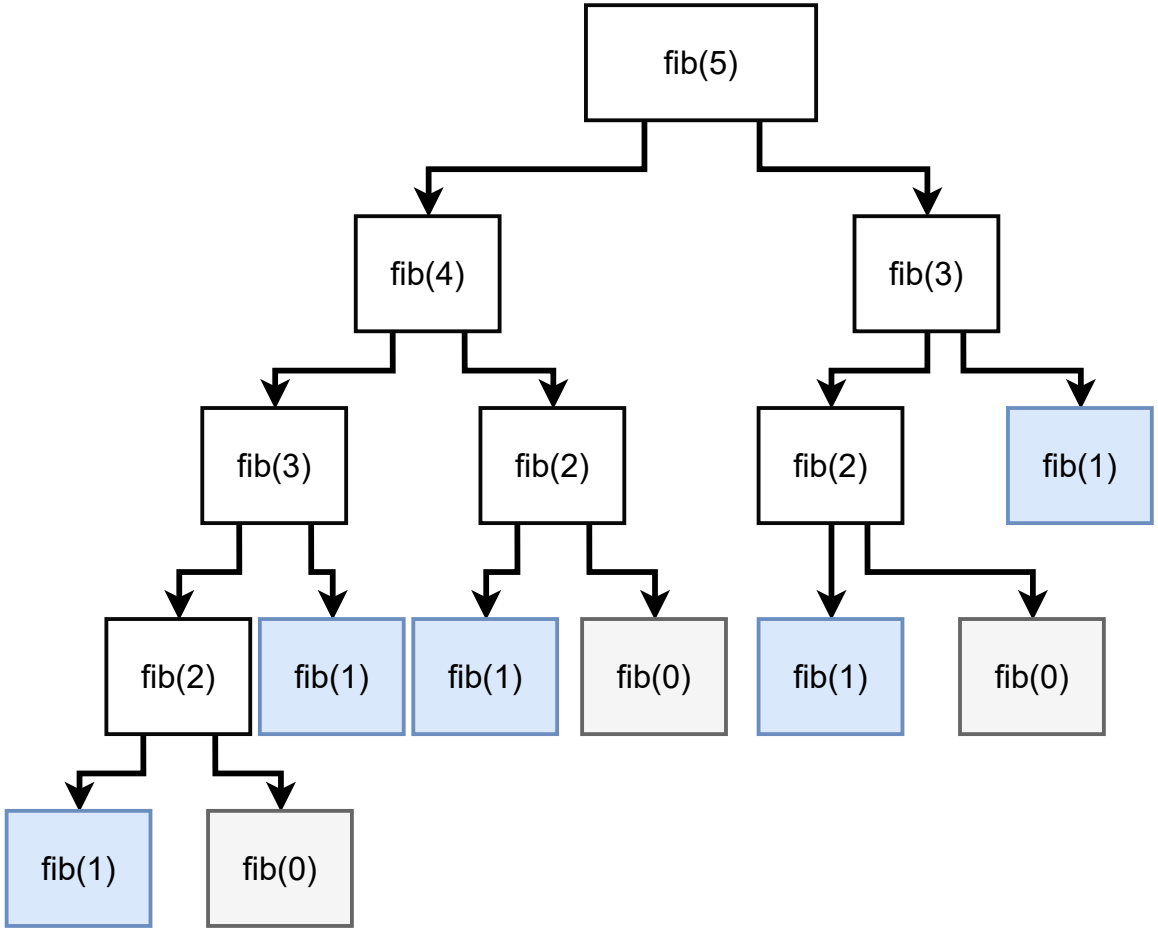
Space Complexity is a thing too

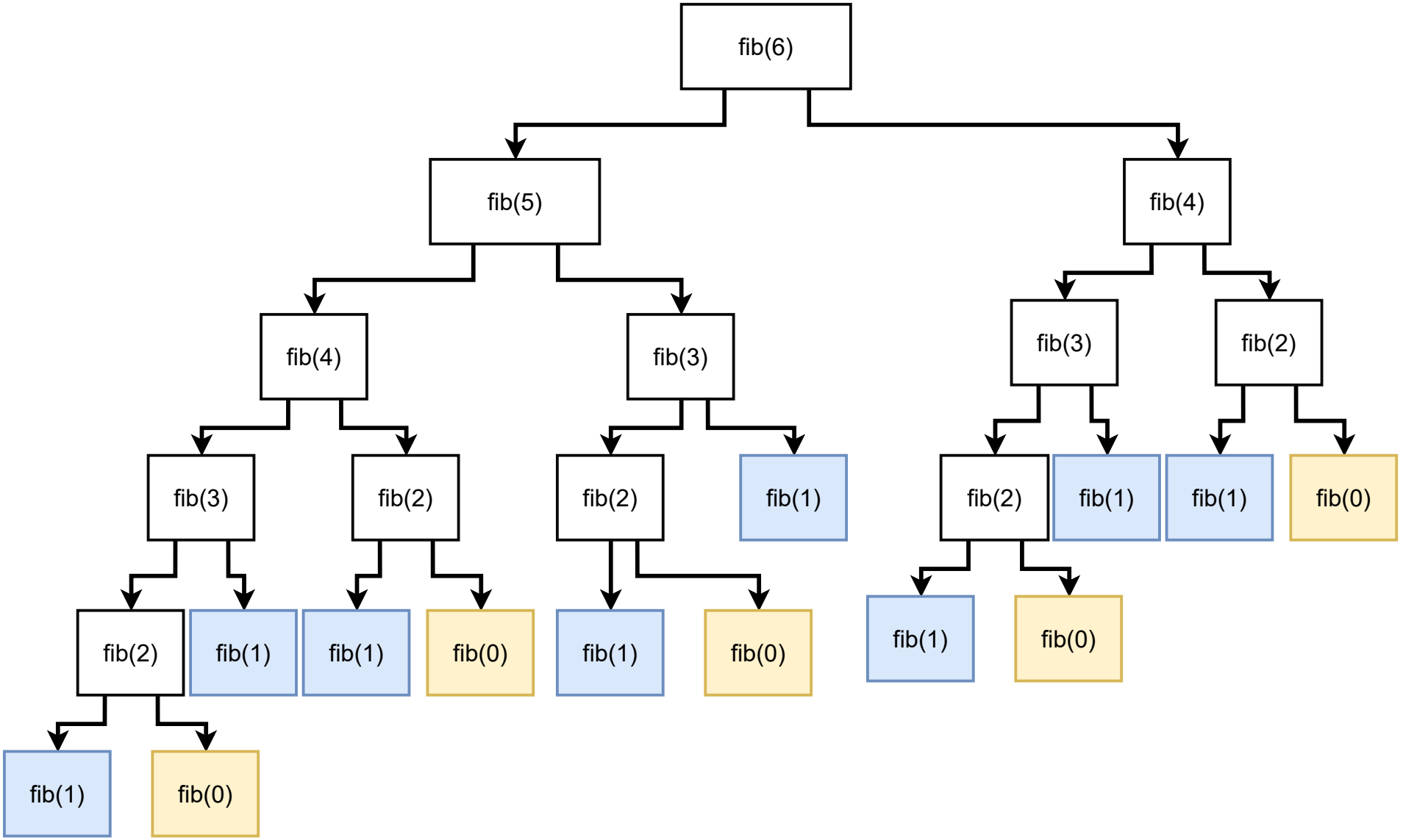
How much more memory is required by doubling the problem set?

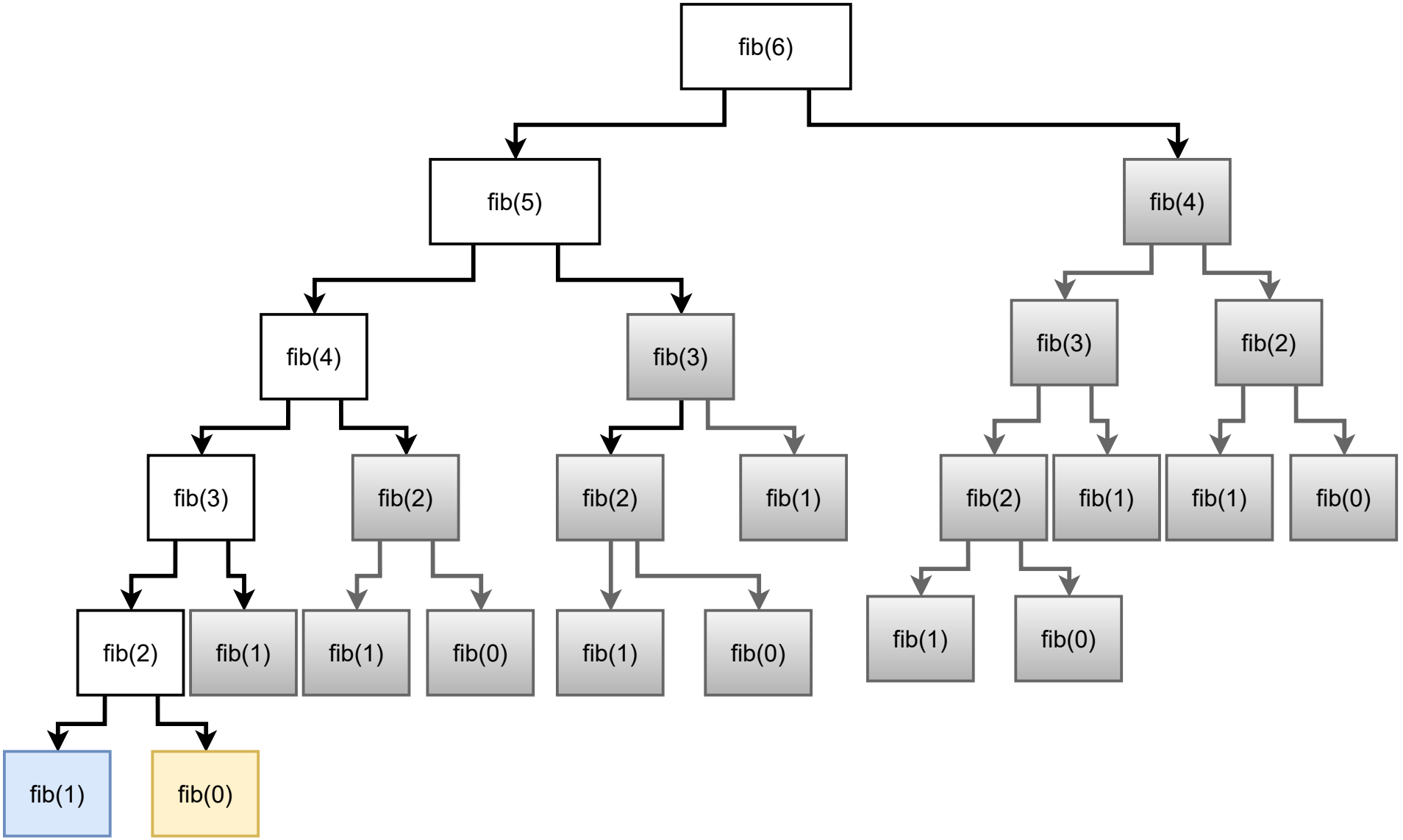
Fibonacci Series




fib(0)
0
fib(1)
1

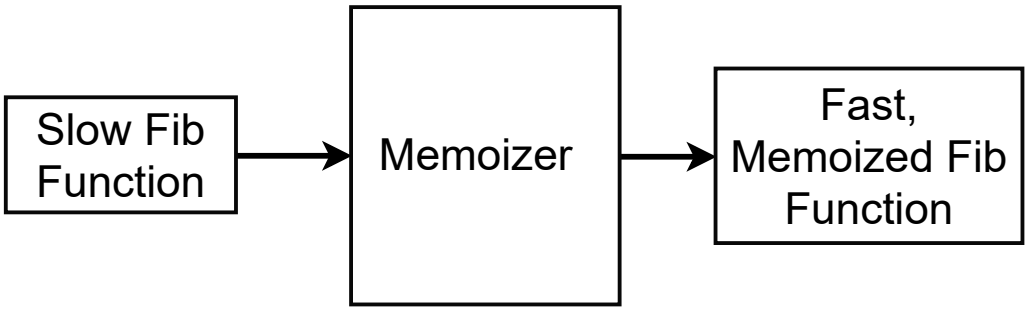

fib(5)
fib(4)
fib(3)
fib(3)
fib(2)
fib(2)
fib(1)
fib(2)
fib(1)
fib(1)
fib(0)
fib(1)
fib(0)
fib(1)
fib(0)


fib(6)
fib(5)
fib(4)
fib(4)
fib(3)
fib(3)
fib(2)
fib(3)
fib(2)
fib(2)
fib(1)
fib(2)
fib(1)
fib(1)
fib(0)
fib(2)
fib(1)
fib(1)
fib(0)
fib(1)
fib(0)
fib(1)
fib(0)


fib(6)
fib(5)
fib(4)
fib(4)
fib(3)
fib(3)
fib(2)
fib(2)
fib(1)
fib(3)
fib(2)
fib(2)
fib(1)
fib(1)
fib(0)
fib(2)
fib(1)
fib(1)
fib(0)
fib(1)
fib(0)
fib(1)
fib(0)
fib(1)
fib(0)

memoization

Store the arguments of each function call along with the result. If the function is called again with the *same arguments*, return the precomputed result, rather than running the function again


Slow Fib
Function
Memoizer
Fast,
Memoized Fib
Function