N-th Fibonacci

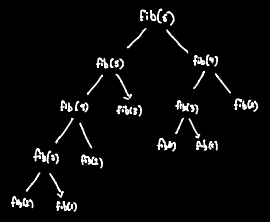
Input: 6
Output: 5

Input: An integer n
Output: The nth Fibonacci number

Binary Recursive

```
// O(2^n) time | O(n) space
function getNthFib(n) {
  if (n === 1) return 0;
  if (n === 2) return 1;

  return getNthFib(n - 1) + getNthFib(n - 2);
}
```



Time: $O(2^n)$ since at each call we do two more fib calls which then have two more fib calls $(2 \times 2 \times 2 \ldots)$

Space: $O(n)$ since the recursive calls use an implicit call stack

Memoization using Hash Map

```
// O(n) time | O(n) space
function getNthFib(n, memoize = { 1: 0, 2: 1 }) {
  if (n in memoize) {
    return memoize[n];
  } else {
    memoize[n] = getNthFib(n - 1, memoize) + getNthFib(n - 2, memoize);
    return memoize[n];
  }
}
```

Time: $O(n)$ since we have to obtain the values of each fib number in order to store it in our memoized hash table thus needing to iterate over n fibonaci's (fib look ups are O(1) time)

Space: $O(n)$ since we store each fib in our memoized hash table AND because we still have our recursive call stack

Multiple Pointers

```
// O(n) time | O(1) space
function getNthFib(n) {
  const lastTwo = [0, 1];
  if (n === 1) return lastTwo[0];
  let counter = 3;
  while (counter <= n) {
    let nextFib = lastTwo[0] + lastTwo[1];
    lastTwo[0] = lastTwo[1];
    lastTwo[1] = nextFib;
    counter++;
  }
  return lastTwo[1];
}
```

Time: $O(n)$ since we have to calculate n fib numbers

Space: $O(1)$ since now there is no recursive call stack and we are only storing the last two fib #'s in our fixed length array