COMP251: DATA STRUCTURES & ALGORITHMS

Instructor: Maryam Siahbani

Computer Information System
University of Fraser Valley

* Some slides from “Java Programming: Program Design Including Data Structures” by Chris Kiekintveld

Recursion

(continue)

Recursion - recap

- Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first

- Recursion is a technique that solves a problem by solving a smaller problem of the same type

- A procedure that is defined in terms of itself

Recursion - recap

- Many methods can be written either with or without using recursion.

  Q: Is the recursive version usually faster?

  A: No -- it's usually slower (due to the overhead of maintaining the stack frames)

  Q: Does the recursive version usually use less memory?

  A: No -- it usually uses more memory (for the stack frames).

  Q: Then why use recursion??

  A: Sometimes it is much simpler to write the recursive version (we'll need to wait until we've discussed trees to see good examples...)

Fibonacci

- Fibonacci can be defined as follows:

$$F_n = F_{n-1} + F_{n-2}$$

$$F_1 = F_2 = 1$$

Fibonacci

- Recursive

```
int fib(int n){
  if (n <= 2)
    return 1;
  return fib(n-1)+fib(n-2);
}
```

Fibonacci

- Recursive

```
int fib(int n){
  if (n <= 2)
    return 1;
  return fib(n-1)+fib(n-2);
}
```

- Iterative

```
int fib (int n) {
  int k1, k2, k3;
  k1 = k2 = k3 = 1;
  for (int j = 3; j <= n; j++)
  {
      k3 = k1 + k2;
      k1 = k2;
      k2 = k3;
  }
  return k3;
}
```

Fibonacci

- Recursive

```
int fib(int n){
  if (n <= 2)
    return 1;
  return fib(n-1)+fib(n-2);
}
```

$$\Theta(??)$$

- Iterative

```
int fib (int n) {
  int k1, k2, k3;
  k1 = k2 = k3 = 1;
  for (int j = 3; j <= n; j++)
  {
      k3 = k1 + k2;
      k1 = k2;
      k2 = k3;
  }
  return k3;
}
```

$$\Theta(n)$$

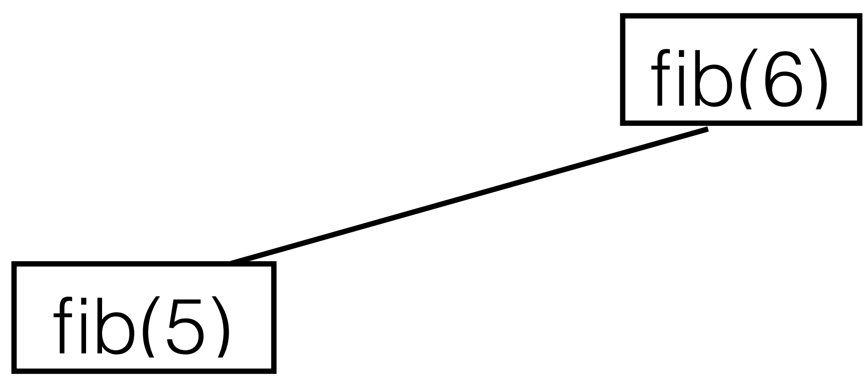
Fibonacci

```
int fib(int n){
  if (n <= 2)
    return 1;
  return fib(n-1)+fib(n-2);
}
```

$\boxed{\text{fib(6)}}$

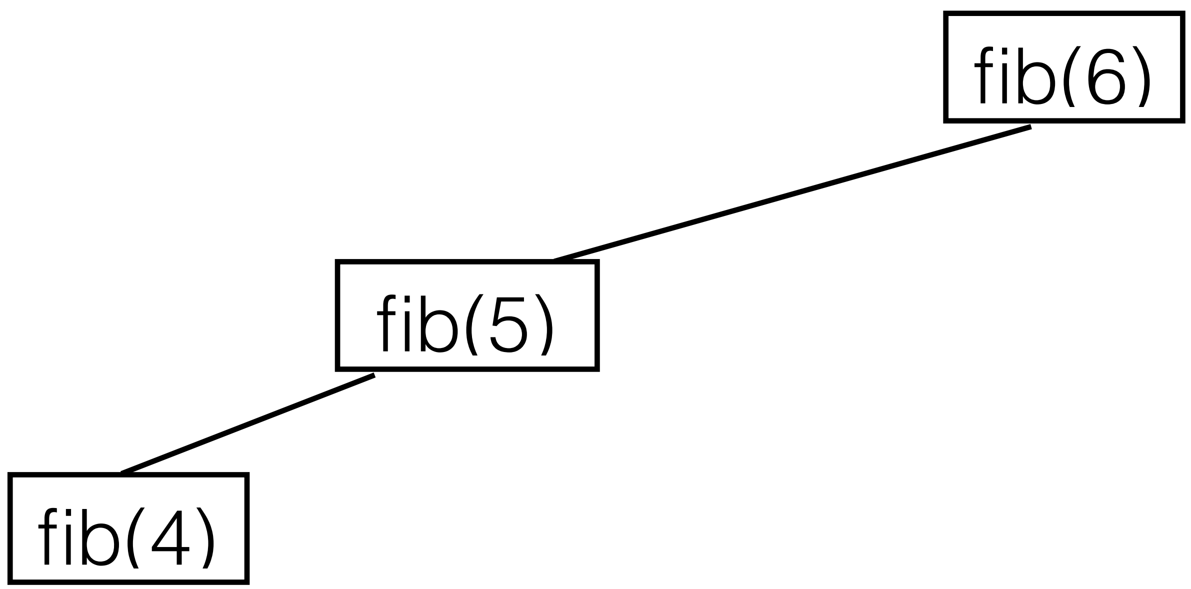
Fibonacci

```
int fib(int n){
  if (n <= 2)
    return 1;
  return fib(n-1)+fib(n-2);
}
```



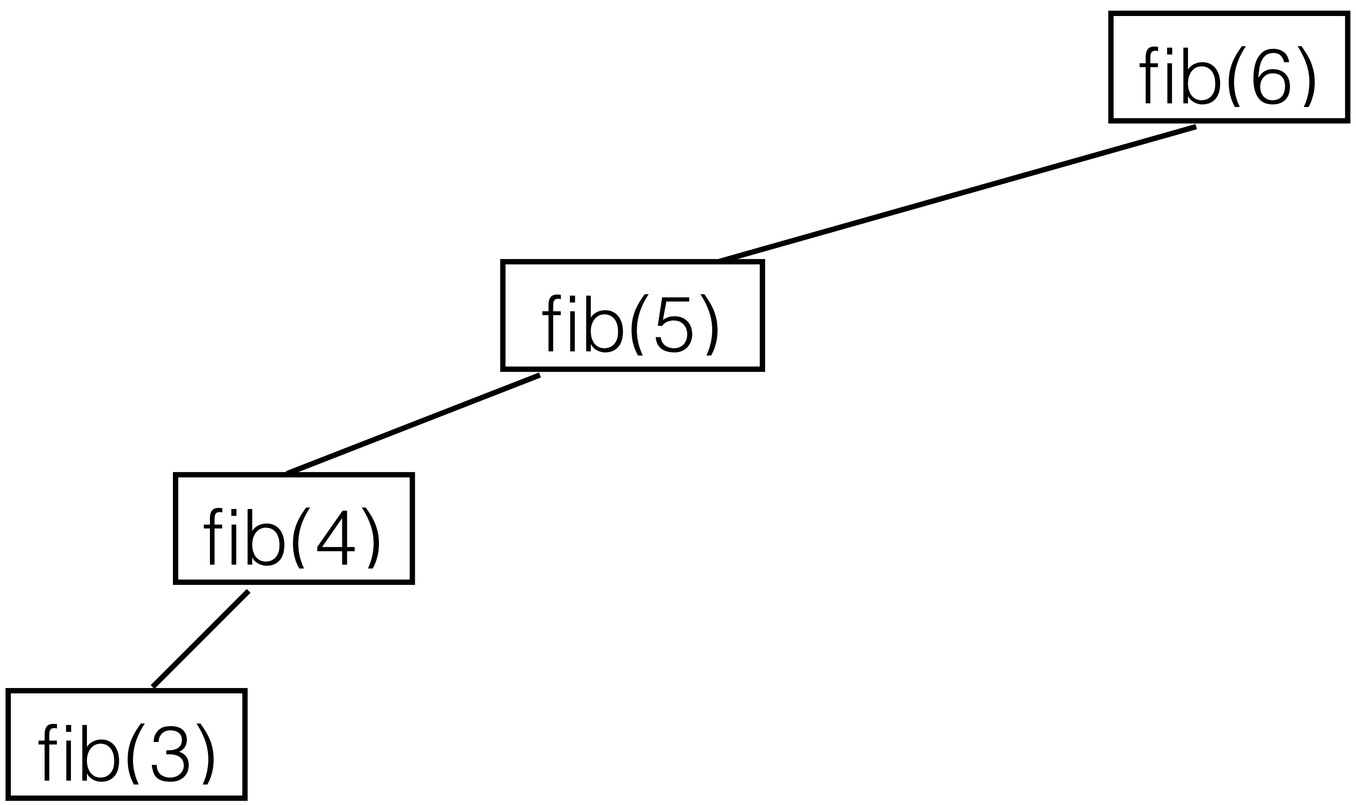
Fibonacci

```
int fib(int n){
  if (n <= 2)
    return 1;
  return fib(n-1)+fib(n-2);
}
```



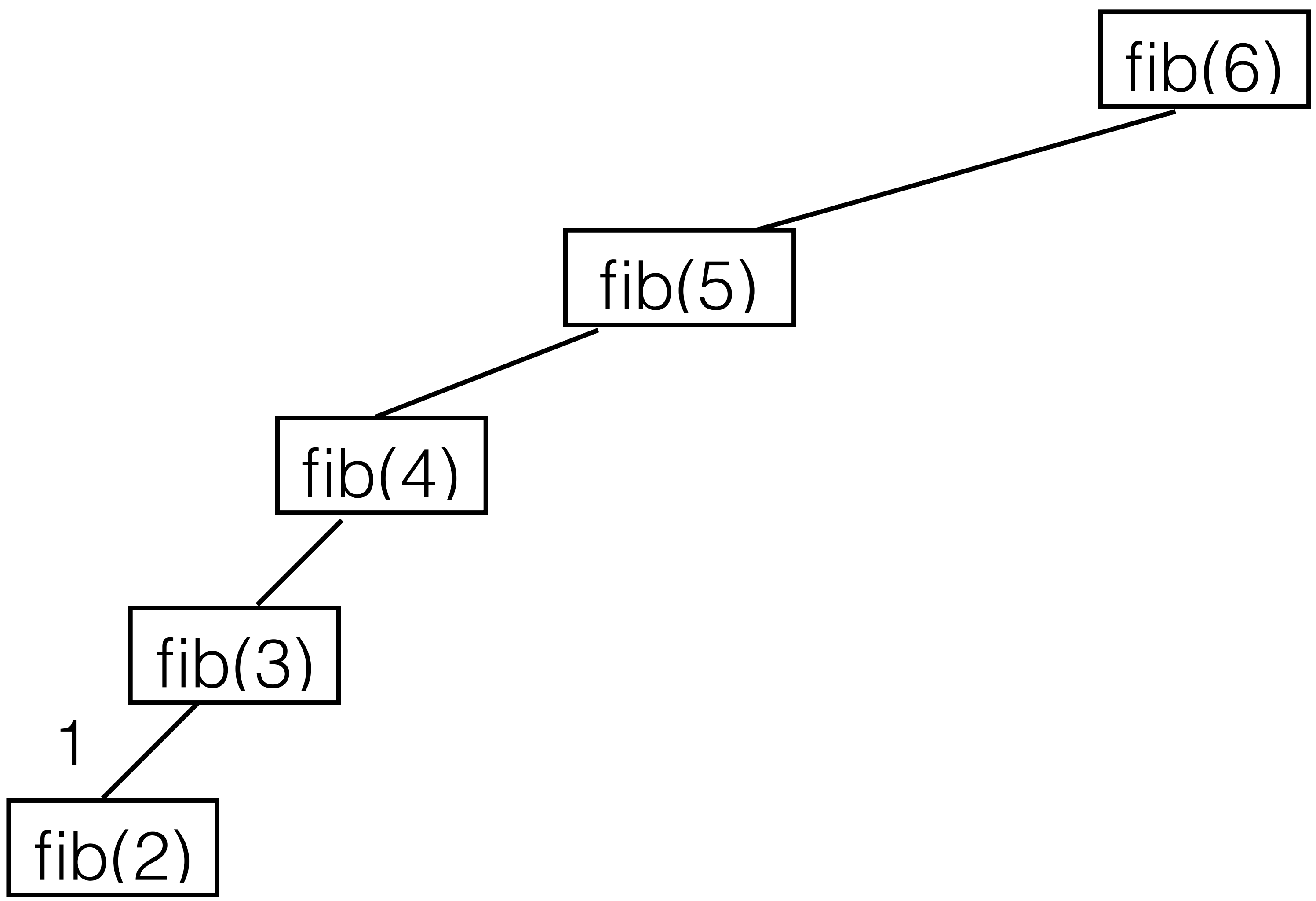
Fibonacci

```
int fib(int n){
  if (n <= 2)
    return 1;
  return fib(n-1)+fib(n-2);
}
```



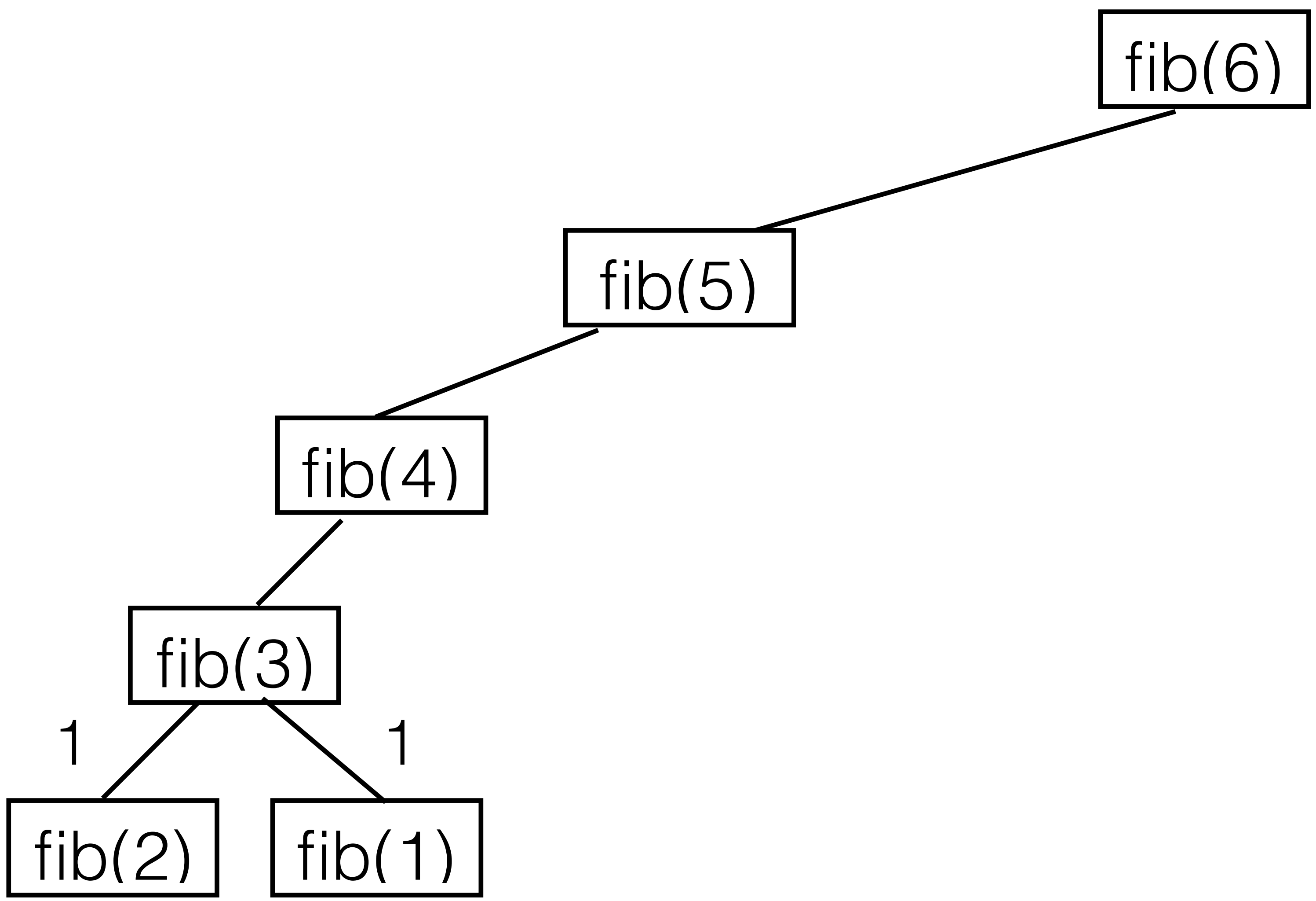
Fibonacci

```
int fib(int n){
  if (n <= 2)
    return 1;
  return fib(n-1)+fib(n-2);
}
```



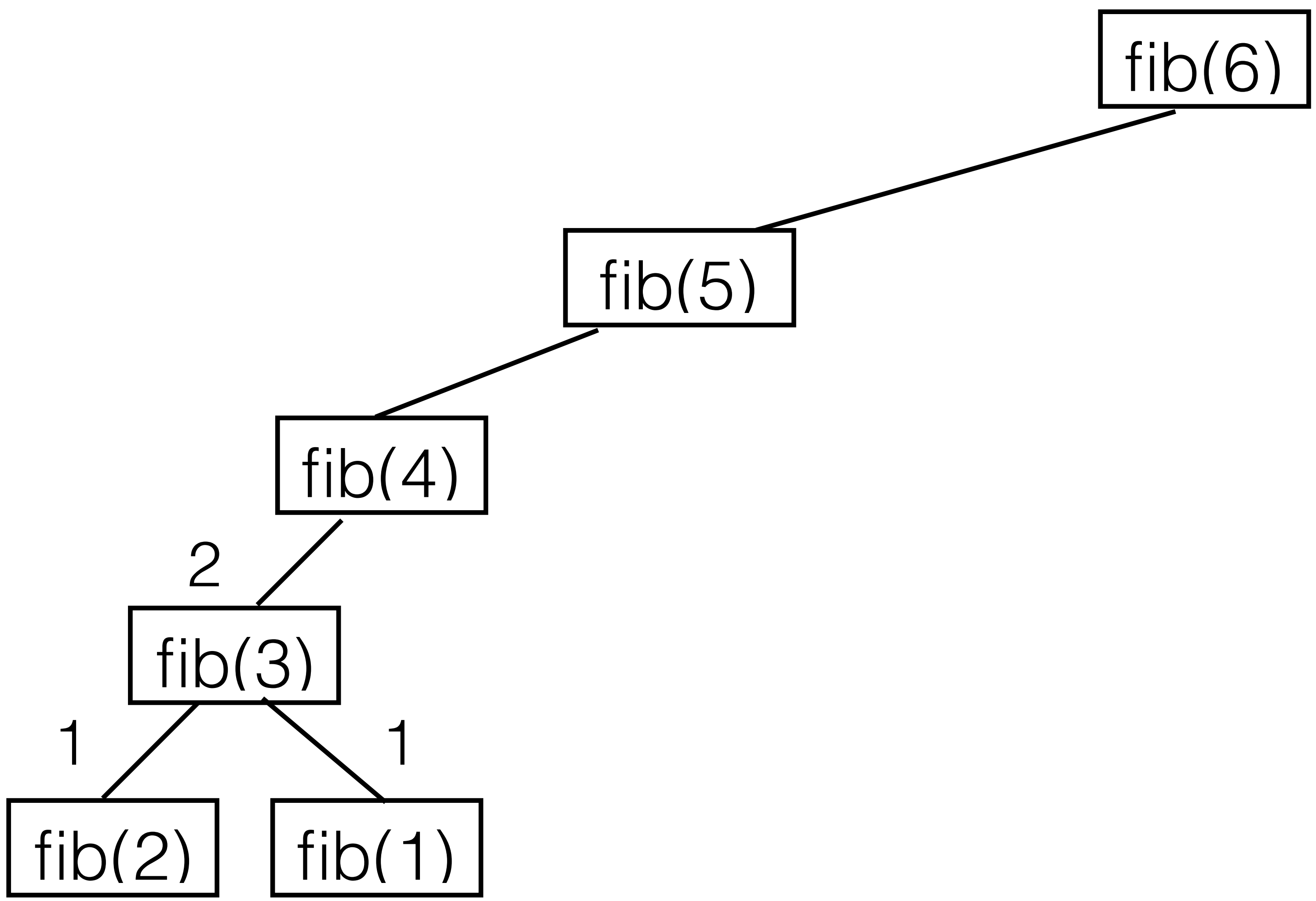
Fibonacci

```
int fib(int n){
  if (n <= 2)
    return 1;
  return fib(n-1)+fib(n-2);
}
```



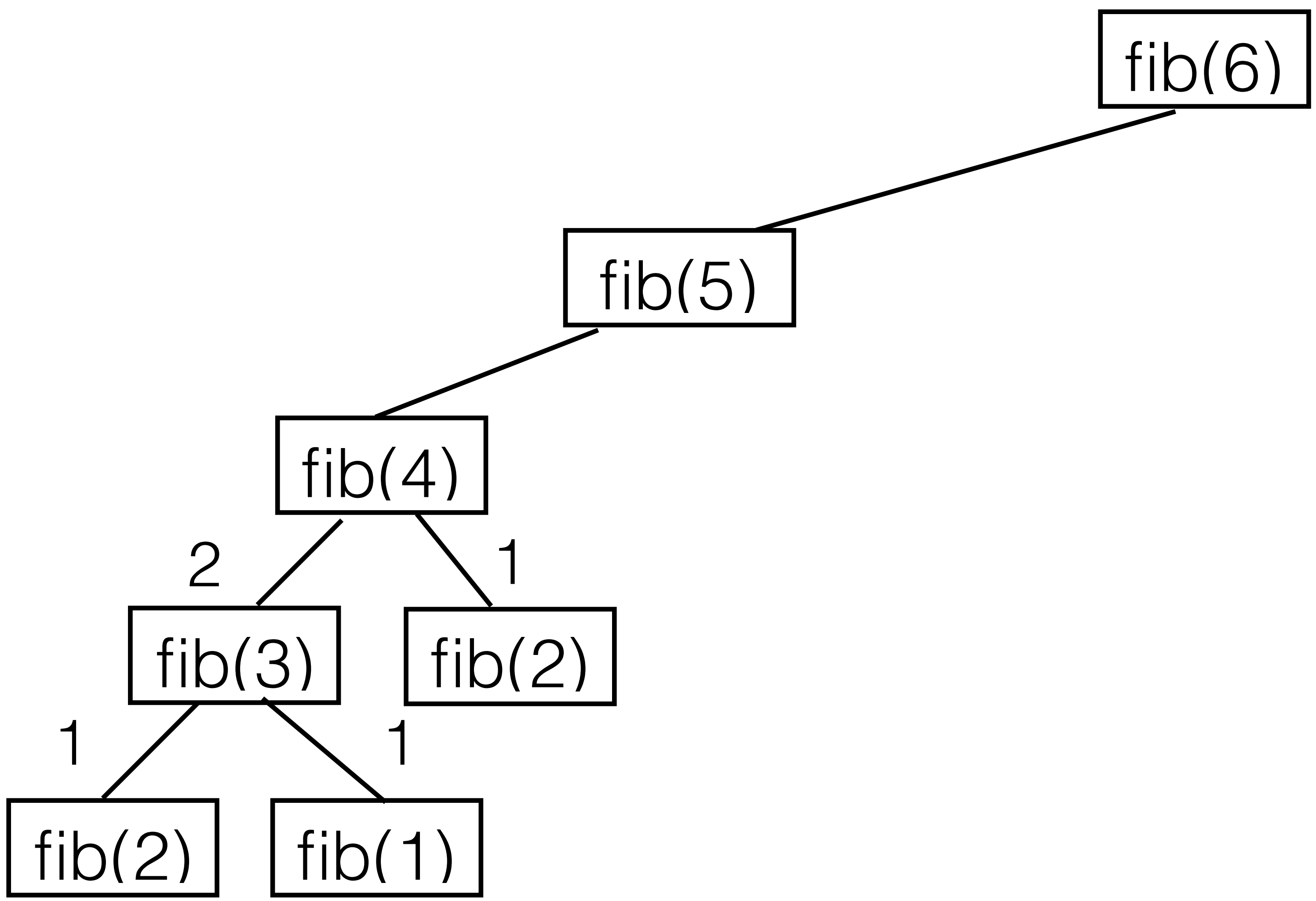
Fibonacci

```
int fib(int n){
  if (n <= 2)
    return 1;
  return fib(n-1)+fib(n-2);
}
```



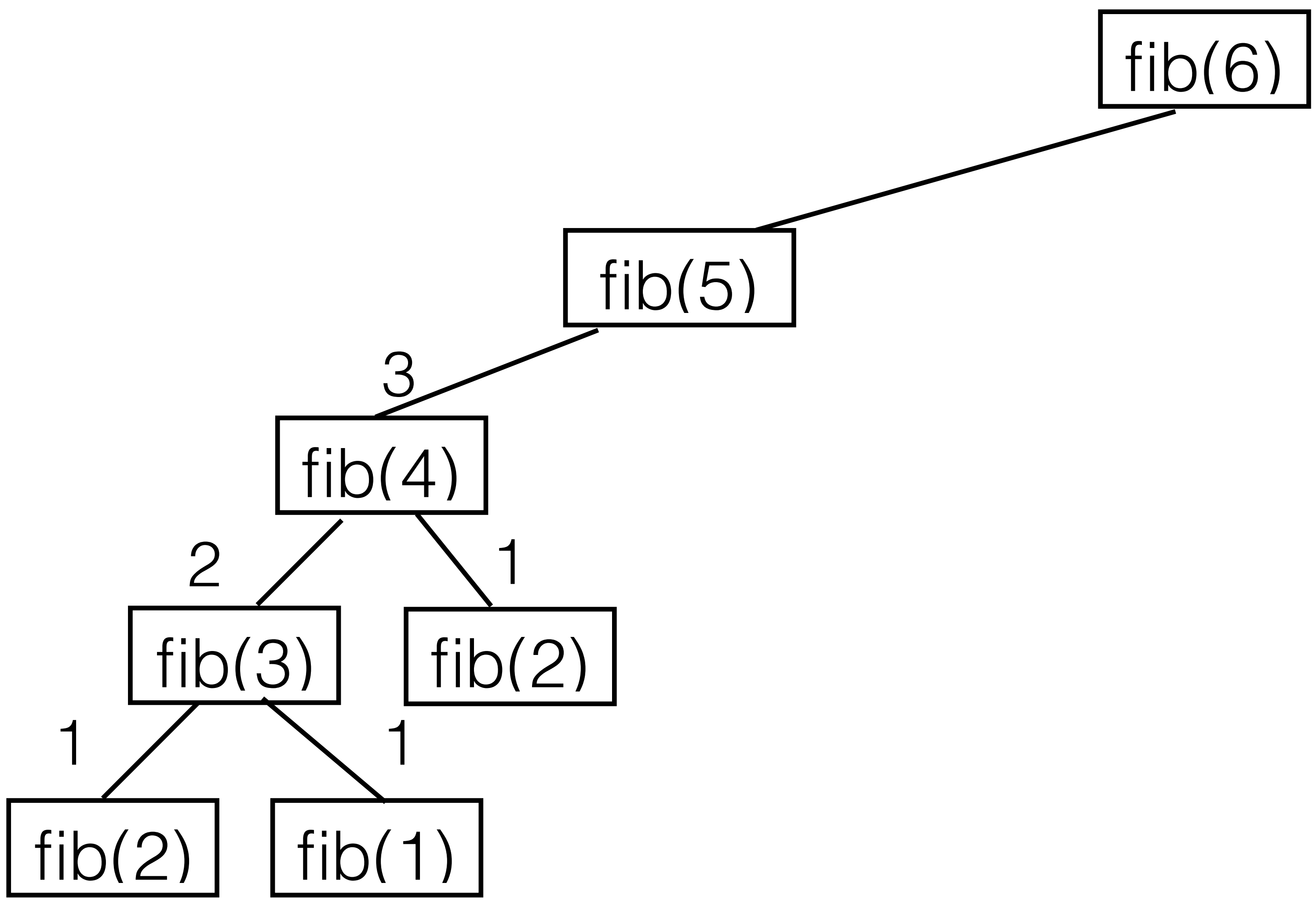
Fibonacci

```
int fib(int n){
  if (n <= 2)
    return 1;
  return fib(n-1)+fib(n-2);
}
```



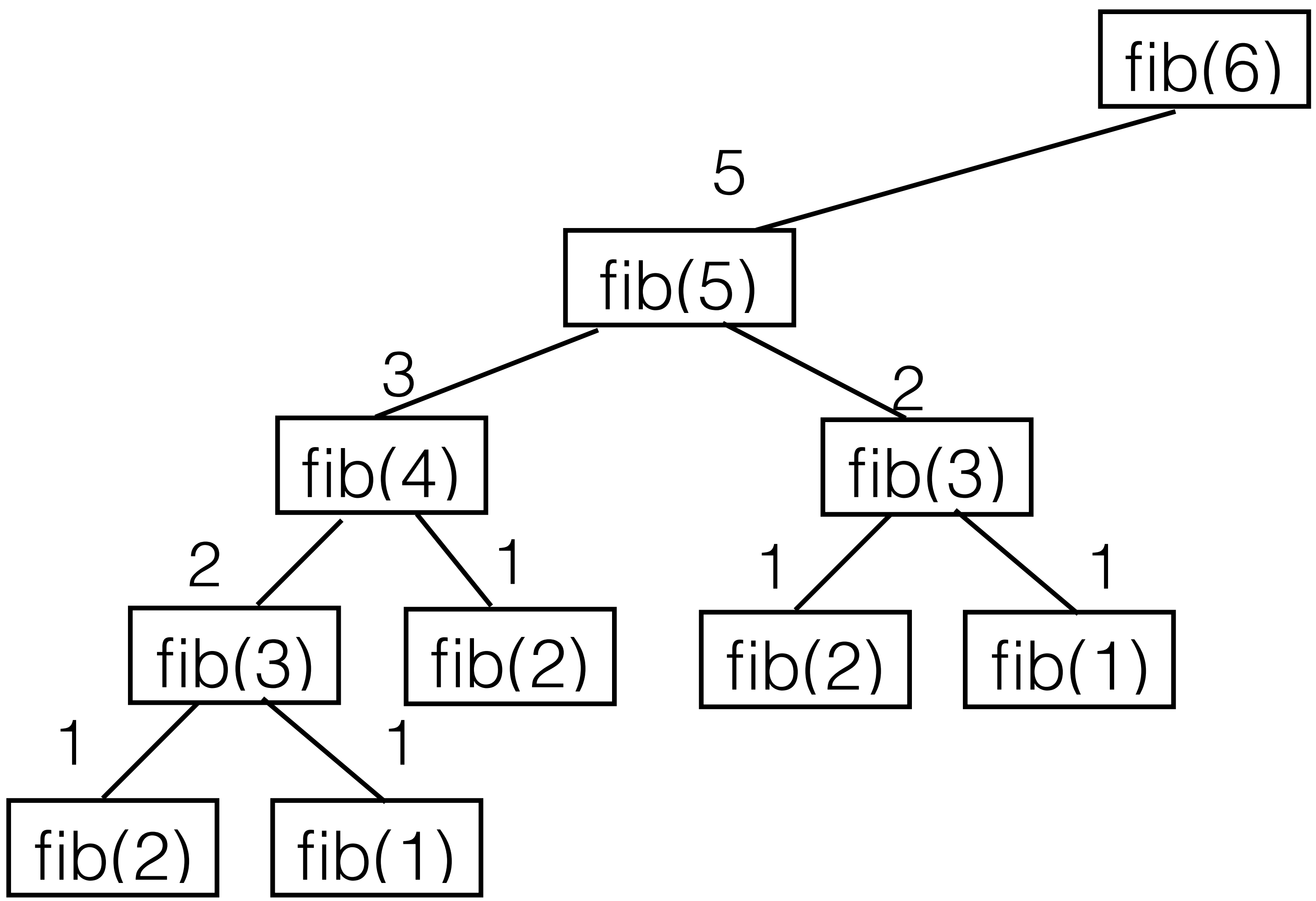
Fibonacci

```
int fib(int n){
  if (n <= 2)
    return 1;
  return fib(n-1)+fib(n-2);
}
```



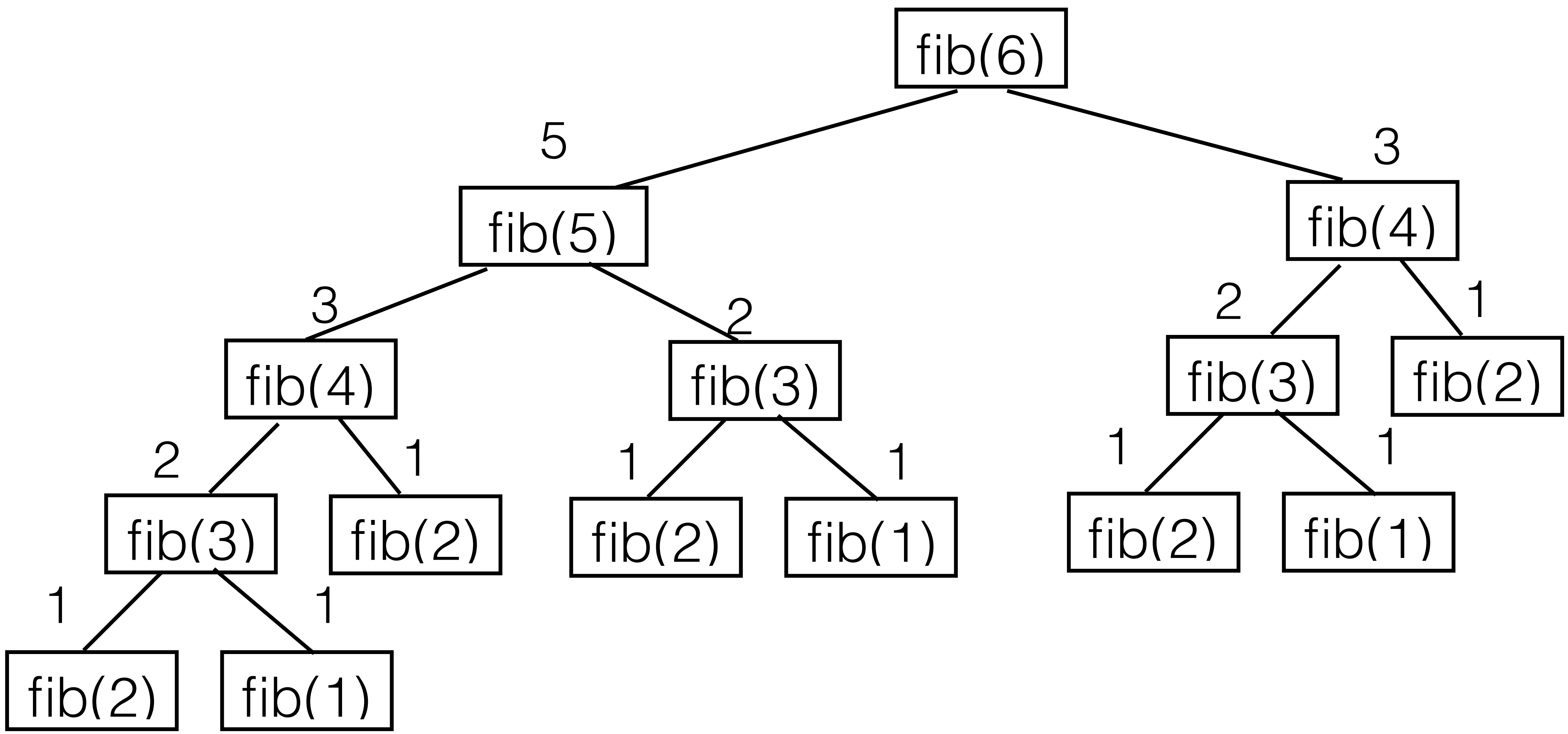
Fibonacci

```
int fib(int n){
  if (n <= 2)
    return 1;
  return fib(n-1)+fib(n-2);
}
```



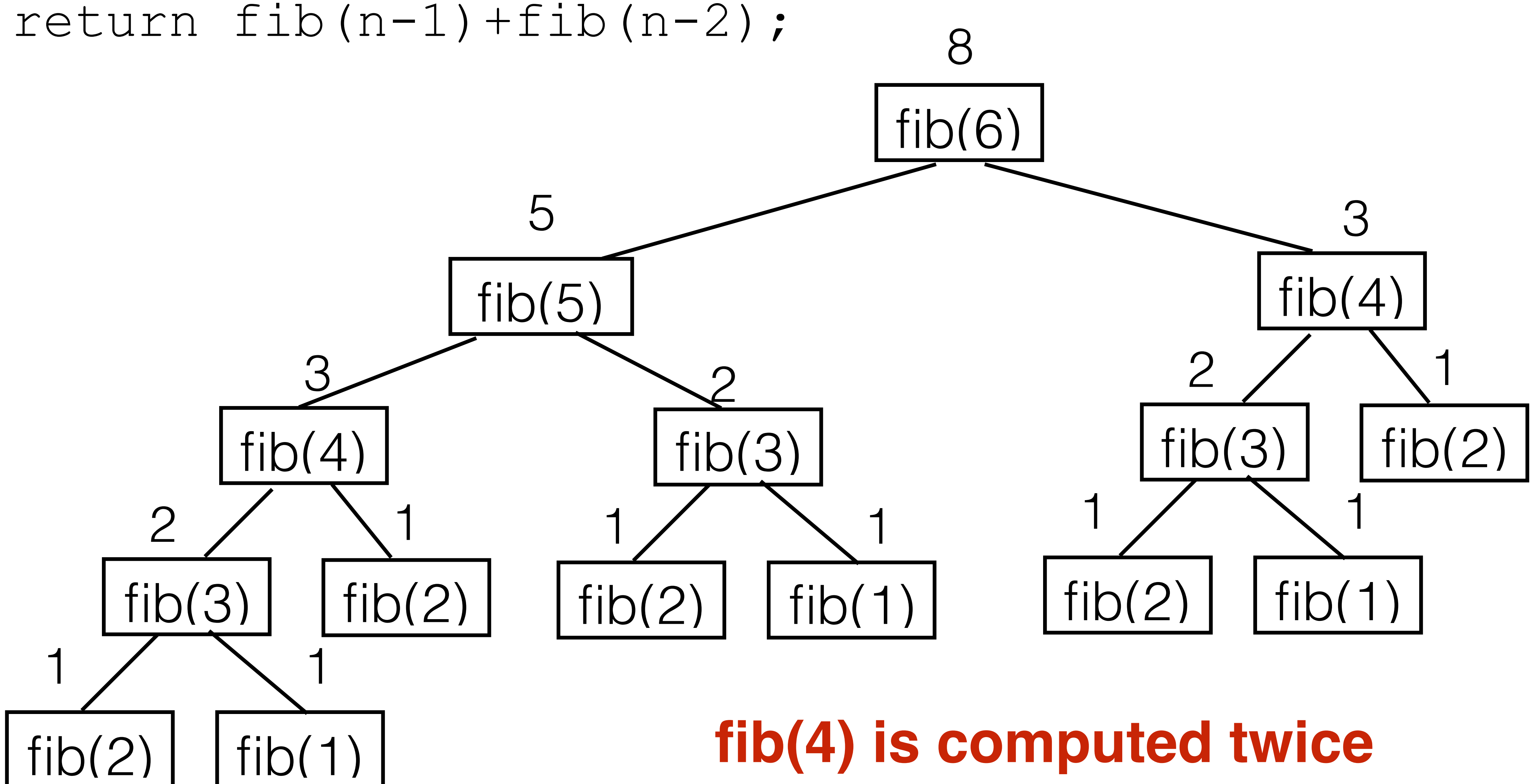
Fibonacci

```
int fib(int n){
  if (n <= 2)
    return 1;
  return fib(n-1)+fib(n-2);
}
```



Fibonacci

```
int fib(int n){
  if (n <= 2)
    return 1;
  return fib(n-1)+fib(n-2);
}
```



fib(4) is computed twice
fib(3) is computed 3 times

Fibonacci

- Recursive

```
int fib(int n){
  if (n <= 2)
    return 1;
  return fib(n-1)+fib(n-2);
}
```

$\Theta(2^n)$

- Iterative

```
int fib (int n) {
  int k1, k2, k3;
  k1 = k2 = k3 = 1;
  for (int j = 3; j <= n; j++)
  {
      k3 = k1 + k2;
      k1 = k2;
      k2 = k3;
  }
  return k3;
}
```

$\Theta(n)$

Fibonacci

- Improved Recursive

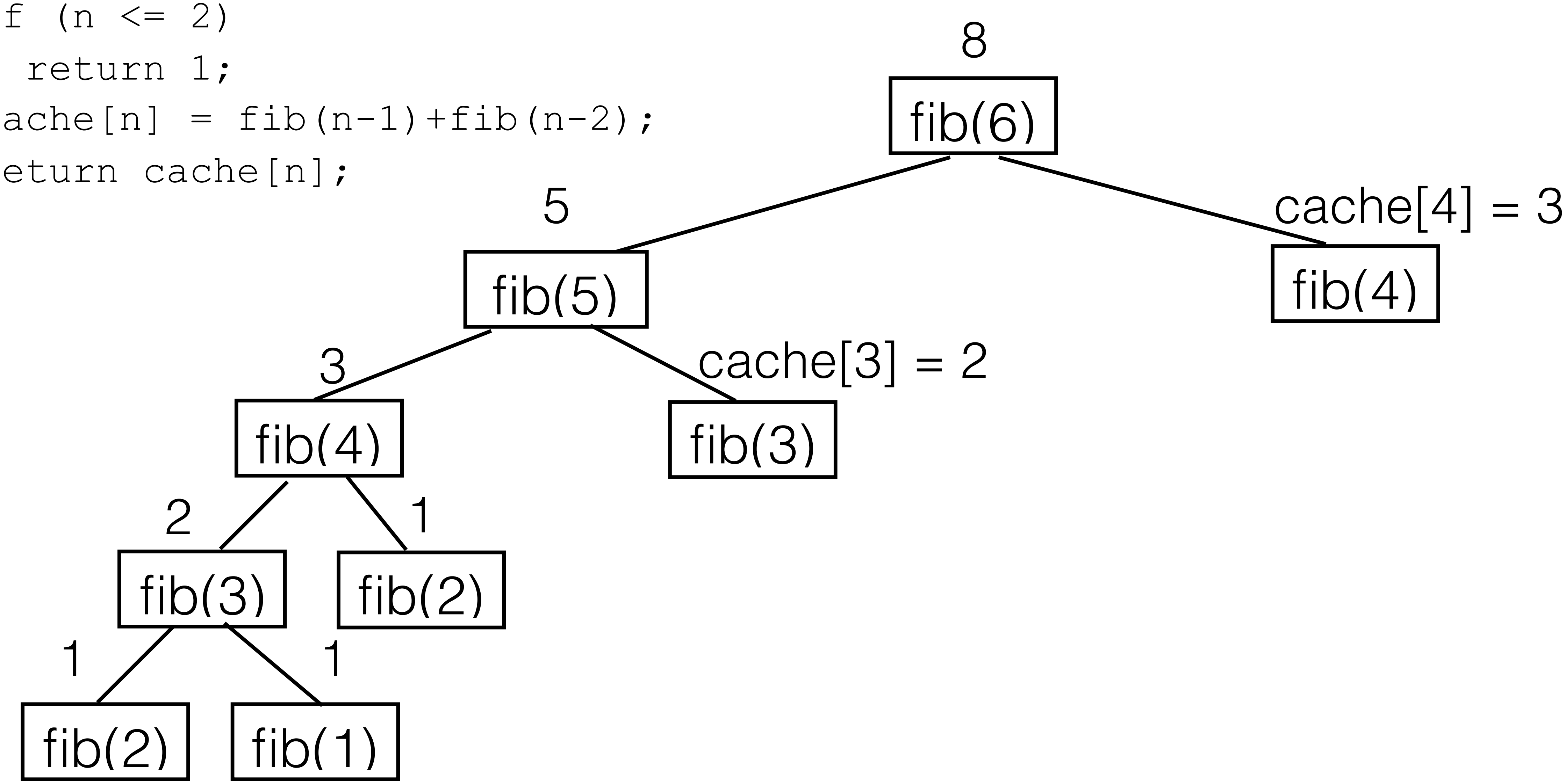
```
int[] cache = new int[MAXSIZE];

int fib(int n){
  if (cache[n] > 0)
    return cache[n];

  if (n <= 2)
    return 1;
  cache[n] = fib(n-1)+fib(n-2);
  return cache[n];
}
```

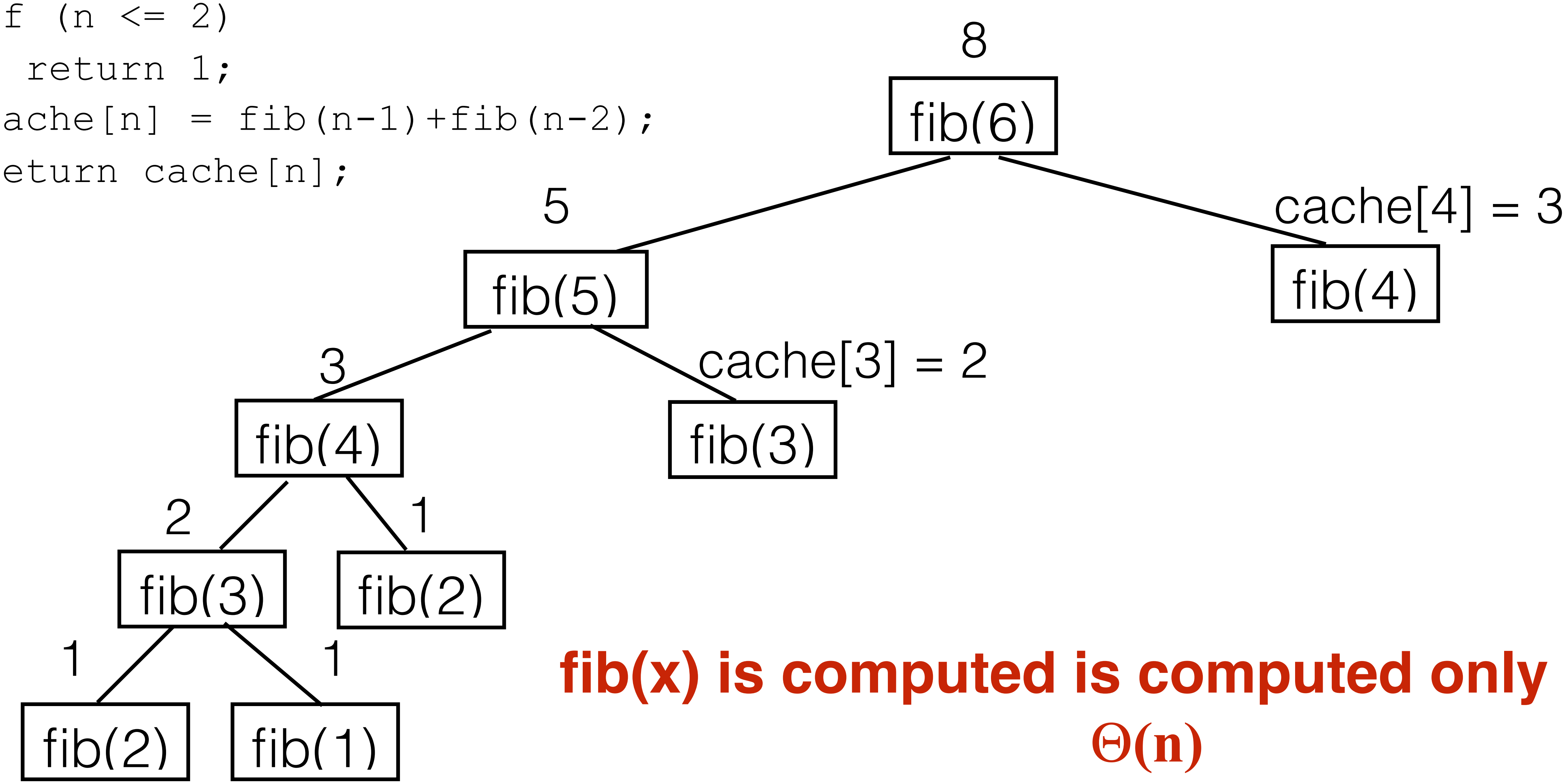
Fibonacci

```
int[] cache = new int[MAXSIZE];
int fib(int n){
  if (cache[n] > 0)
      return cache[n];
  if (n <= 2)
    return 1;
  cache[n] = fib(n-1)+fib(n-2);
  return cache[n];
}
```



Fibonacci

```
int[] cache = new int[MAXSIZE];
int fib(int n){
  if (cache[n] > 0)
     return cache[n];
  if (n <= 2)
    return 1;
  cache[n] = fib(n-1)+fib(n-2);
  return cache[n];
}
```



fib(x) is computed is computed only once

$\Theta(n)$

Correctness

- We can use mathematical induction to prove the correctness of recursive algorithms

- In math, when we use induction to prove a theorem, we need to show:

  1. that the base case (usually n=0 or n=1) is true

  2. that case k implies case k+1 (if case k is correct we can prove case k+1 is correct)

- We can apply similar approach to prove correctness of recursive algorithms

Example - Factorial

- Let's prove the correctness of the recursive version of factorial.

```
public int factorial(int n){
      if (n==0)
         return(1);
      else
         return(n * f(n-1));
}
```

Example - Factorial

- We need to prove:

  1. the base case: factorial(0) = 0!
     - The correctness of the factorial method for n=0 is obvious from the code: when n==0 it returns 1.

Example - Factorial

- We need to prove:

1. the base case: factorial(0) = 0!
    - The correctness of the factorial method for n=0 is obvious from the code: when n==0 it returns 1.
2. k implies k+1: if factorial(k) = k!, then factorial(k+1)=(k+1)!
    - Looking at the code, we see for n != 0, factorial(n) = (n)*factorial(n-1).
    - So factorial(k+1) = (k+1)*factorial(k)
    - By assumption, factorial(k) = k!
    - factorial(k+1) = (k+1)*(k!)   => factorial(k+1) returns (k+1)!

Example - Factorial

- We need to prove:

1. the base case: factorial(0) = 0!
   - The correctness of the factorial method for n=0 is obvious from the code: when n==0 it returns 1.

2. k implies k+1: if factorial(k) = k!, then factorial(k+1)=(k+1)!
   - Looking at the code, we see for n != 0, factorial(n) = (n)*factorial(n-1).
   - So factorial(k+1) = (k+1)*factorial(k)
   - By assumption, factorial(k) = k!
   - factorial(k+1) = (k+1)*(k!)   => factorial(k+1) returns (k+1)!

The proof is just valid for n >= 0!

Search

- Design a method that returns `true` if element `n` is a member of `array x[]` and `false` if not

Search

- Design a method that returns `true` if element `n` is a member of `array x[]` and `false` if not

- Iterative approach

```
public boolean search(int[] x, int n) {
    for(int i = 0; i < x.length, i++) {
        if (x[i] == n]) return true;
    }
    return false;
}
```

Search

- Design a method that returns `true` if element `n` is a member of `array x[]` and `false` if not

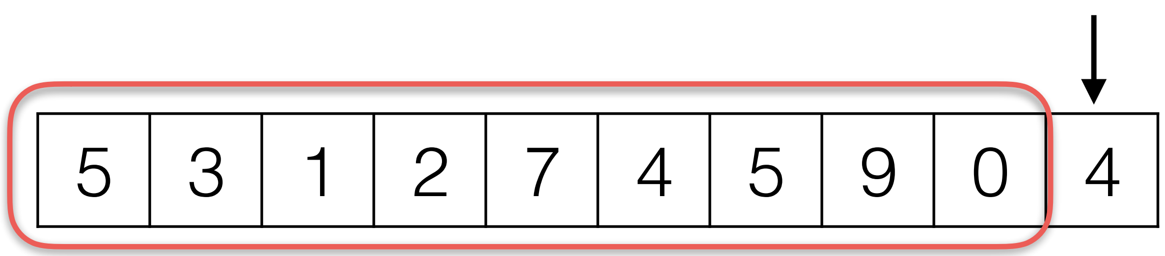
- Recursive

5	3	1	2	7	4	5	9	0	4

Search

- Design a method that returns `true` if element `n` is a member of `array x[]` and `false` if not

- Recursive



  - `n` is in the last cell (`x[size-1] == n`)
  - or `n` is in the rest of array, which can be seen as a smaller array of length: size-1 (`search(size-1,n) == true`)

Search

- Design a method that returns `true` if element `n` is a member of `array x[]` and `false` if not

- Recursive

```
boolean search(int[] x, int size, int n) {
    if (size > 0) {
        if (x[size-1] == n)
            return true;
        else
            return search(x, size-1, n);
    }
    return false;
}
```

Search

- The problem: these methods are slow, $\Theta(n)$
- Recall the phone book example
- "Linear search" – need to look at every element
- "Binary search" is much faster on sorted data

Binary Search

```
search(phonebook, name)
    if only one page
        scan for the name
    else
        open to the middle
        determine if name is before or after this page

        if name is before
            search (first half of phonebook, name)
        else
            search (second half of phonebook, name)
```

Binary Search

```
boolean binarySearch(int[] x, int start, int end, int n)
{
  if (end < start) return false;
  int mid = (start+end) / 2;
  if (x[mid] == n)
    return true;

  if (x[mid] < n)
     return search(x, mid+1, end, n);
  else
     return search(x, start, mid-1, n);
}
```