ICT1008 Data Structures and Algorithms

Lecture 3: Recursion, Greedy Algorithms

Agenda

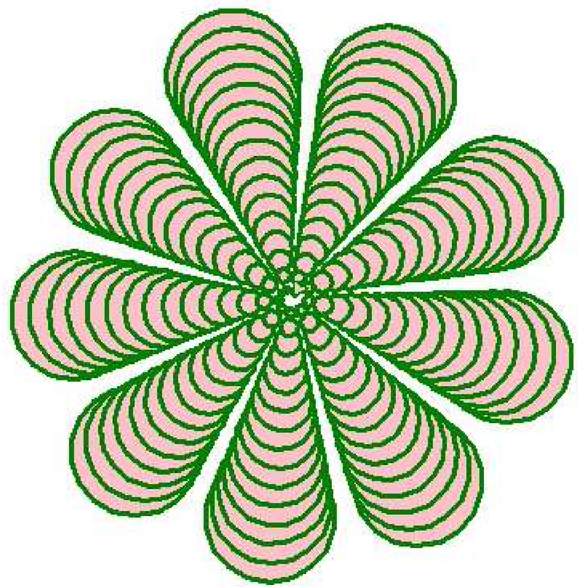
- Basic Recurrence
- Recursive Algorithms
- Analysis of Recursive Algorithms
- Greedy Algorithms

Recommended Readings

1. Runestone Interactive book: “Problem Solving with Algorithms and Data Structures Using Python”
   – Section “Recursion”

What is recursion?

"**Recursion** is a method of solving problems that involves breaking a problem down into smaller and smaller subproblems until you get to a small enough problem that it can be solved trivially. Usually recursion involves a function calling itself. While it may not seem like much on the surface, recursion allows us to write elegant solutions to problems that may otherwise be very difficult to program." [1]



Recurrence

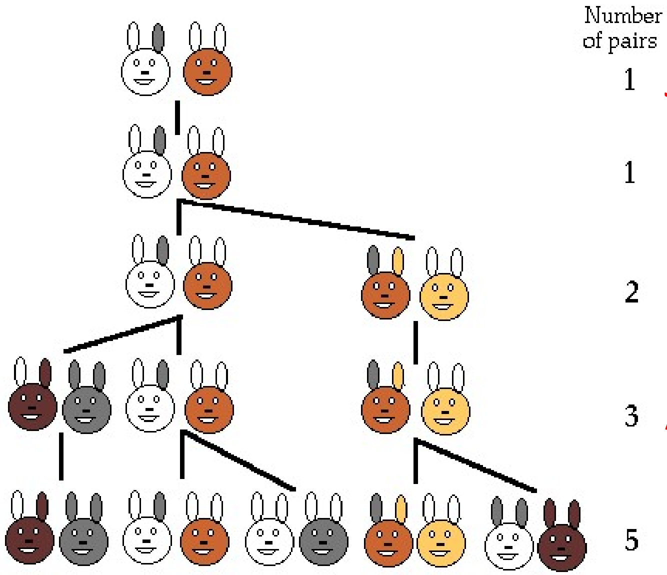
Recursion is based on a mathematical concept called Recurrence

- Recursive Function
  - A function that calls itself
  - A way to terminate itself through a base case

Fibonacci Sequence

- Leonardo Fibonacci (Mathematician) asked a question involving the reproduction of a single pair of rabbits which is the basis of the Fibonacci sequence.

- Suppose a newly born pair of rabbits (a male and female) are put in a field.
- Rabbits are able to mate at the age of one month so that at the end of second month a female can produce another pair of rabbits.
- Rabbits never die and the female always produces a new pair every month from the second month on.
- How many pairs will there be in one year?



Jan
Feb
Mar
Apr
May
:
Dec

Ans: 144

Recurrence relations: Fibonacci sequence

Fibonacci sequence:

$$f_2 = f_1 + f_0 = 1 + 0 = 1$$
$$f_3 = f_2 + f_1 = 1 + 1 = 2$$
$$f_4 = f_3 + f_2 = 2 + 1 = 3$$
$$f_5 = f_4 + f_3 = 3 + 2 = 5$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34…

$$f_n = \overbrace{(f_{n-1} + f_{n-2})} \text{ for } n \geq 2$$

$$\left.\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \end{aligned}\right\} \text{initial condition}$$

Recurrence relation: an equation that relates the n^{th} element f_n of a sequence to some of its predecessors $f_0, f_1, \ldots, f_{n-1}$.

We need an `initial condition` that provides the starting values for a finite number of elements of the sequence.

Recursive calls

- Used frequently in computer programs.
- A recursive function calls itself.

<u>Example</u>

The factorial function:

$n! \ = \ 1 \times 2 \times 3 \cdots (n-1) \times n$ for $n \geq 1.$

$0! \ = \ 1$ by definition.

Hence, $n! \ = \ n \times (n-1)!$ for $n \geq 1.$

$$factorial(n) \ = \ n \ \times \ factorial(n-1)$$

Example: $N!$

```
def factorial(n):
    if n==0:
        answer = 1
    else:
        answer = n*factorial(n-1)
    return answer
```

A test for the BASE CASE (initial condition) enables the recursive calls to stop.

A shorter version that does exactly the same thing:

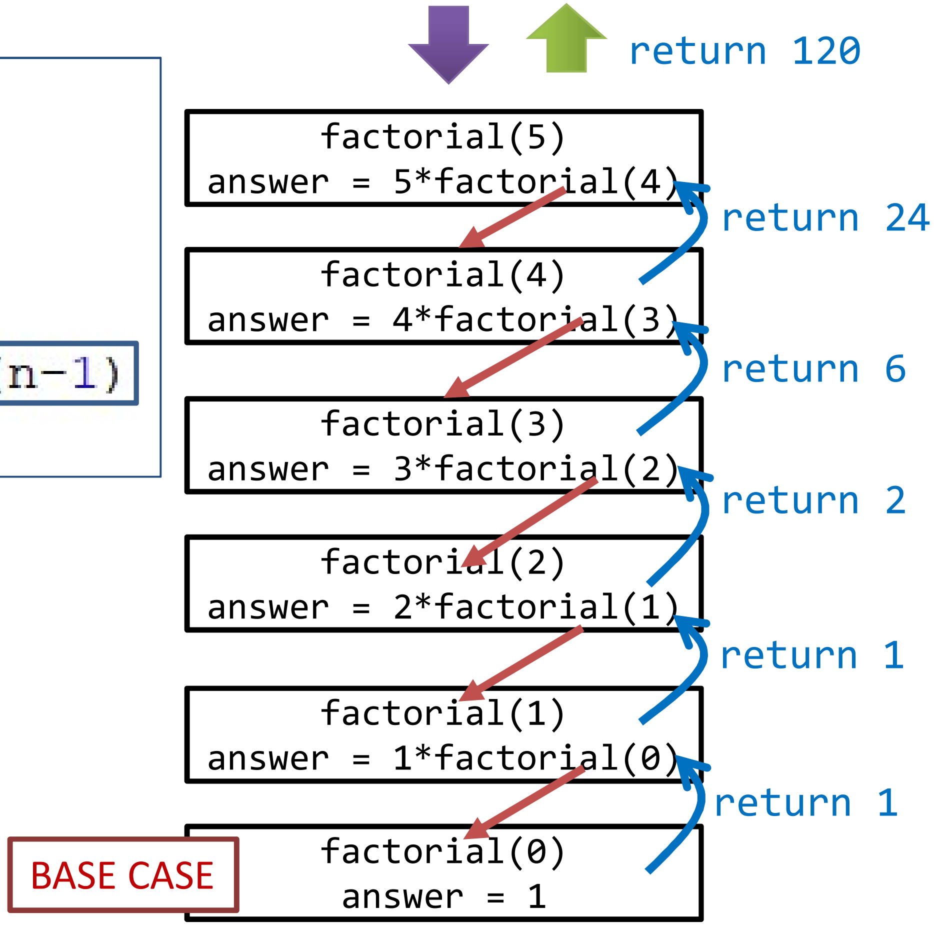
```
def factorial(n):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)
```

Each recursive call solves an identical (but smaller) problem.

Anatomy of a recursive call

```
def factorial(n):
    if n==0:
        answer = 1
    else:
        answer = n*factorial(n-1)
    return answer
```

Remember that each call to a function starts that function anew. That means it has its own copy of any local values, including the values of the parameters.



Example: Time Analysis for N!

$factorial(n) = n \times factorial(n-1)$

Let $T(n)$ be the number of multiplications needed to compute $factorial(n)$.

$T(n) = T(n-1) + 1$
$T(0) = 1$

$T(n-1)$ multiplications are needed to compute $factorial(n-1)$, and one more multiplication is needed to multiply the result by n.

Example: Time Analysis for N!

Using the method of `backward substitutions`

$$
\begin{aligned}
T(n) &= T(n-1) + 1 \\
&= [T(n-2) + 1] + 1 \\
&= T(n-2) + 2 \\
&= [T(n-3) + 1] + 2 \\
&= T(n-3) + 3 \\
&\dots \\
&= [T(n-n) + 1] + (n-1) \\
&= T(n-n) + n \\
&= T(0) + n \\
&= 1 + n
\end{aligned}
$$

Time efficiency of the recursive n! algorithm is of $O(n)$.

Beauty of Recursive Algorithms

$$f(0) = 1$$
$$f(n) = n \times factorial(n-1), \mathrm{n} \geq 1$$

Direct translation between the recurrence relation and the recursive algorithm

```
def factorial(n):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)
```

Strategy for Designing Recursive Algorithm

1. Identify the recurrence relation to solve the problem.
2. Translate the recurrence relation to a recursive algorithm.
3. Take note to translate the initial condition in the recurrence relation into the BASE CASE for the recursive algorithm.

Example: Fibonacci sequence

Recall that:

$$f_n = f_{n-1} + f_{n-2}, \qquad n \geq 2$$

$$f_0 = 0,\ f_1 = 1\ (initial\ condition)$$

```python
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

BASE CASE

Recursive Game

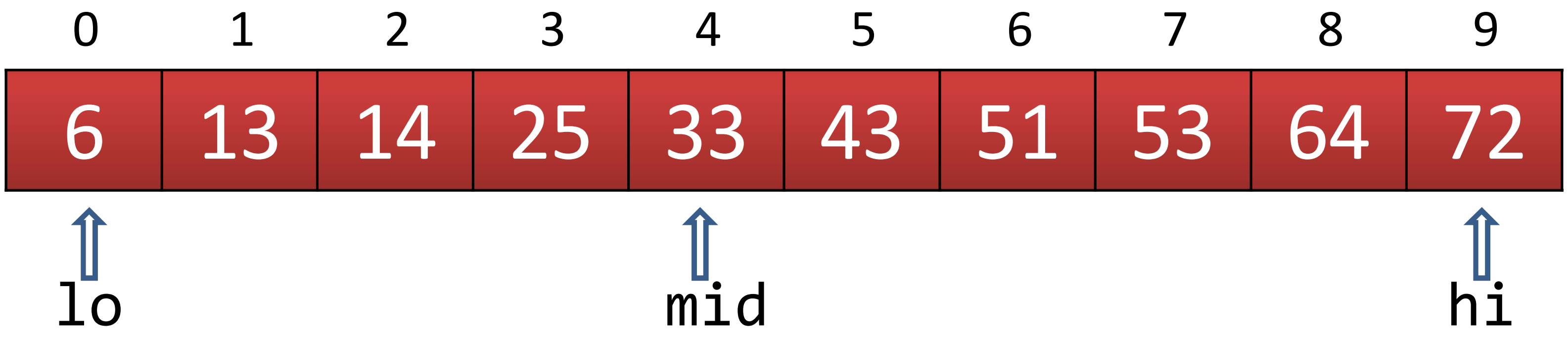
http://www.softschools.com/games/logic_games/tower_of_hanoi/

Example: Binary Search

- Goal.
  Given a sorted array and a key.
  Find index (location) of the key in the array.

- Binary search.
  Compare key against middle entry.
  1. Smaller, search in the left half.
  2. Bigger, search in the right half.
  3. <u>Equal, return the index.</u>
  4. <u>Size <= 0, return -1.</u>

Example: Binary Search




```python
def search(a,lo,hi,key):
    if lo>hi: return -1

    mid = (int)((hi+lo)/2)
    if a[mid]>key:
        return search(a,lo,mid-1,key)
    elif a[mid]<key:
        return search(a,mid+1,hi,key)
    else:
        return mid
```

Binary search.
Compare key against middle entry.

1. Smaller, search in the left half.
2. Bigger, search in the right half.
3. Equal, return the index.
4. Size <= 0, return -1.

Example: Exponentiation

- Compute a^n for an integer n.
- A quick and easy algorithm.

```
def power(a,n):
    answer = 1
    for i in range(n):
        answer = answer * a
    return answer
```

Time complexity
$O(n)$

- 2^8 = 2x2x2x2x2x2x2x2.
- Faster way to compute a^n ?

Example: FAST Exponentiation

- Compute a^n for an integer n.
- Divide and conquer strategy.

$$2^8 = 2^4 \times 2^4 = 16 \times 16 = 256$$

$$2^4 = 2^2 \times 2^2 = 4 \times 4 = 16$$

$$2^2 = 2 \times 2 = 4$$

$$\boxed{a^n = \begin{cases} a^{n/2}(a^{n/2}) & \text{if } n \text{ is even} \\ a^{n/2}(a^{n/2})(a) & \text{if } n \text{ is odd} \end{cases}}$$

Example: FAST Exponentiation

$$a^n = \begin{cases} a^{n/2}(a^{n/2}) & \text{if } n \text{ is even} \\ a^{n/2}(a^{n/2})(a) & \text{if } n \text{ is odd} \end{cases}$$

```
def power(a,n):
    if n==0: return 1
    answer = power(a,(int)(n/2))
    if n%2 == 0:
        return answer*answer
    else:
        return answer*answer*a
```

Note: It is important that we use the variable $answer$ twice instead of calling the function $power(a, n)$ twice.

Analysis of Recursive Algorithms

1. Decide on parameter n indicating $input\ size$.
2. Identify algorithm's $basic\ operation$.
3. Set up a $recurrence\ relation$ with an appropriate $initial\ condition$ expressing the number of times the basic operation is executed.
4. Solve the recurrence (or, at least, establish the solution's $order\ of\ growth$) by backward substitutions or other methods.

Example: FAST Exponentiation analysis

```
def power(a,n):
    if n==0: return 1
    answer = power(a,(int)(n/2))
    if n%2 == 0:
        return answer*answer
    else:
        return answer*answer*a
```

Let $T(n)$ be the runtime of the algorithm $power(a, n)$.

$$T(n) = 1 + T(n/2)$$
$$T(0) = 1$$

We need to solve the recurrence relation $T(n)$.

Example: FAST Exponentiation analysis

$$T(n) = 1 + T(n/2)$$

$$T(0) = 1$$

Use the method of backward substitutions:

$$T(n) = 1 + T\left(\frac{n}{2}\right)$$

$$= 1 + \left(1 + T\left(\frac{n}{2^2}\right)\right) = 2 + T\left(\frac{n}{2^2}\right) = 3 + T\left(\frac{n}{2^3}\right)$$

$$\ldots$$

assume $n = 2^k$

$$= (k+1) + T\left(\frac{n}{(2^{k+1})}\right) = (k+1) + T\left(\frac{2^k}{(2^{k+1})}\right)$$

$$= (k+1) + T(0) = k + 2$$

Example: FAST Exponentiation analysis

$T(n) = k + 2$ where $n = 2^k$;

therefore $k = \lg n$.

We have:

$T(n) = \lg n + 2 = O(\lg n)$

Example: FAST Exponentiation analysis

```python
def power(a,n):
    if n==0: return 1
    answer = power(a,(int)(n/2))
    if n%2 == 0:
        return answer*answer
    else:
        return answer*answer*a
```

Time efficiency of the recursive power algorithm is of $O(\lg n)$.

Example: Multiplication

Calculate the product x * y.

Approach 1:

- Add y repeatedly for x times. Requires O(x) additions.
- Divide and conquer strategy?

```
x = x0;
y = y0;
z = 0;
while x > 0, {
    z = z + y;
    x = x - 1;
}
```

Example: FAST Multiplication

Calculate the product x * y.

Approach 2:

- Left-shift
  - multiply by 2
- Right-shift
  - divide by 2
- Runtime behaviour
  T(x) = O(lg x)
  - reduces time complexity from O(x) to O(lg x)

```
x = x0;
y = y0;
z = 0;
While x > 0 {
  if odd(x) {
     z = z + y;
     x = x - 1;
  }
  x = x/2;
  y = 2*y;
}
```

Example: FAST Multiplication

Numerical Example

$x = 9,$	$y = 17,$	$z = 0$
$z = 0 + 17 = 17;$	$x = 9 - 1 = 8;$	
$x = 8/2 = 4;$	$y = 2*17 = 34;$	
$x = 4/2 = 2;$	$y = 2*34 = 68;$	
$x = 2/2 = 1;$	$y = 2*68 = 136;$	
$z = 17 + 136 = 153;$	$x = 1 - 1 = 0;$	

```
x = x0;
y = y0;
z = 0;
While x > 0 {
  if odd(x) {
     z = z + y;
     x = x - 1;
  }
  x = x/2;
  y = 2*y;
}
```

O(lg x) divisions and multiplications.

Recursive vs. Iterative

One should be careful with recursive algorithms because their conciseness, clarity and simplicity may hide their inefficiencies.

Example: Fibonacci Sequence

Recall that:

$$f_n = (f_{n-1} + f_{n-2}), \qquad n \geq 2$$

$$f_0 = 0, \quad f_1 = 1 \ \ (initial\ condition)$$

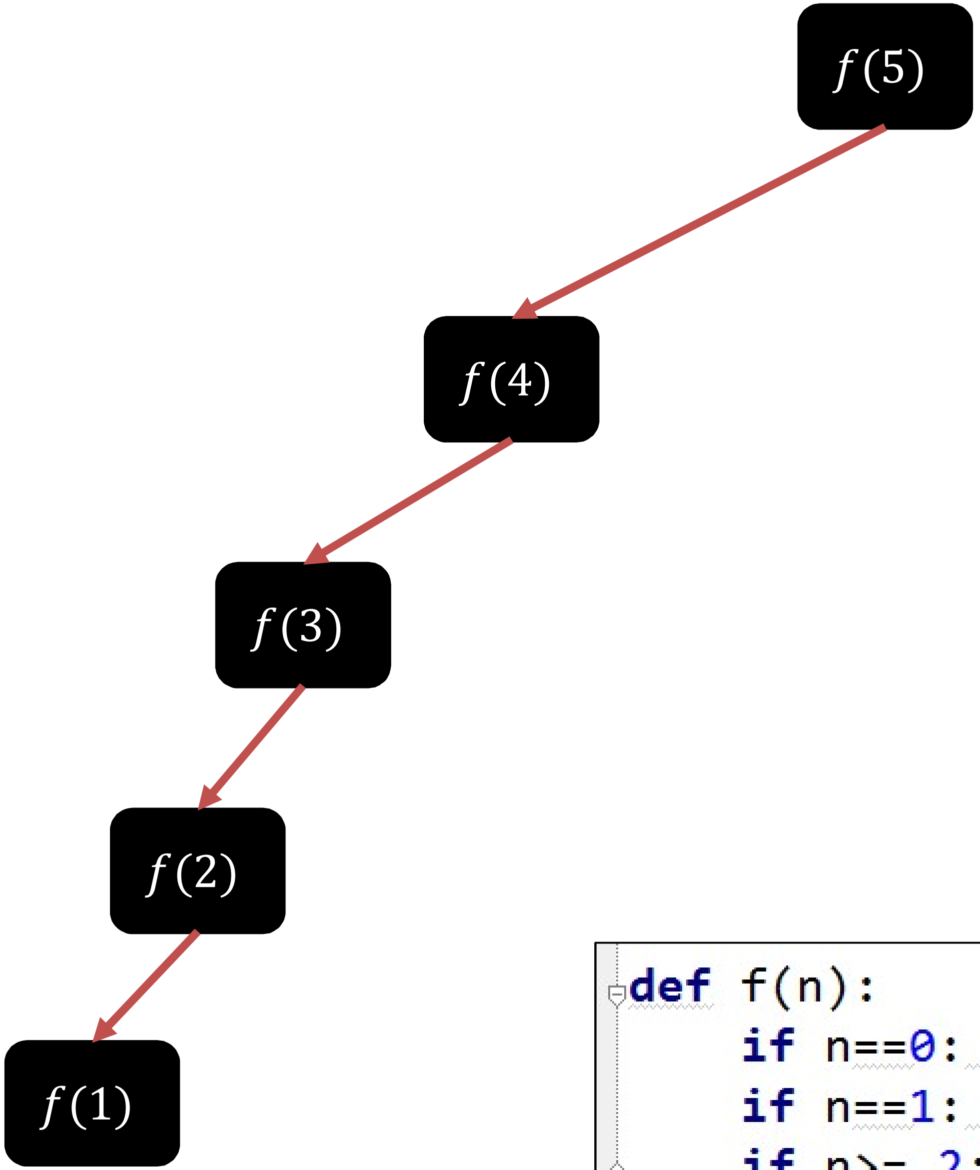
$\Updownarrow$

```python
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

Example: Fibonacci Sequence

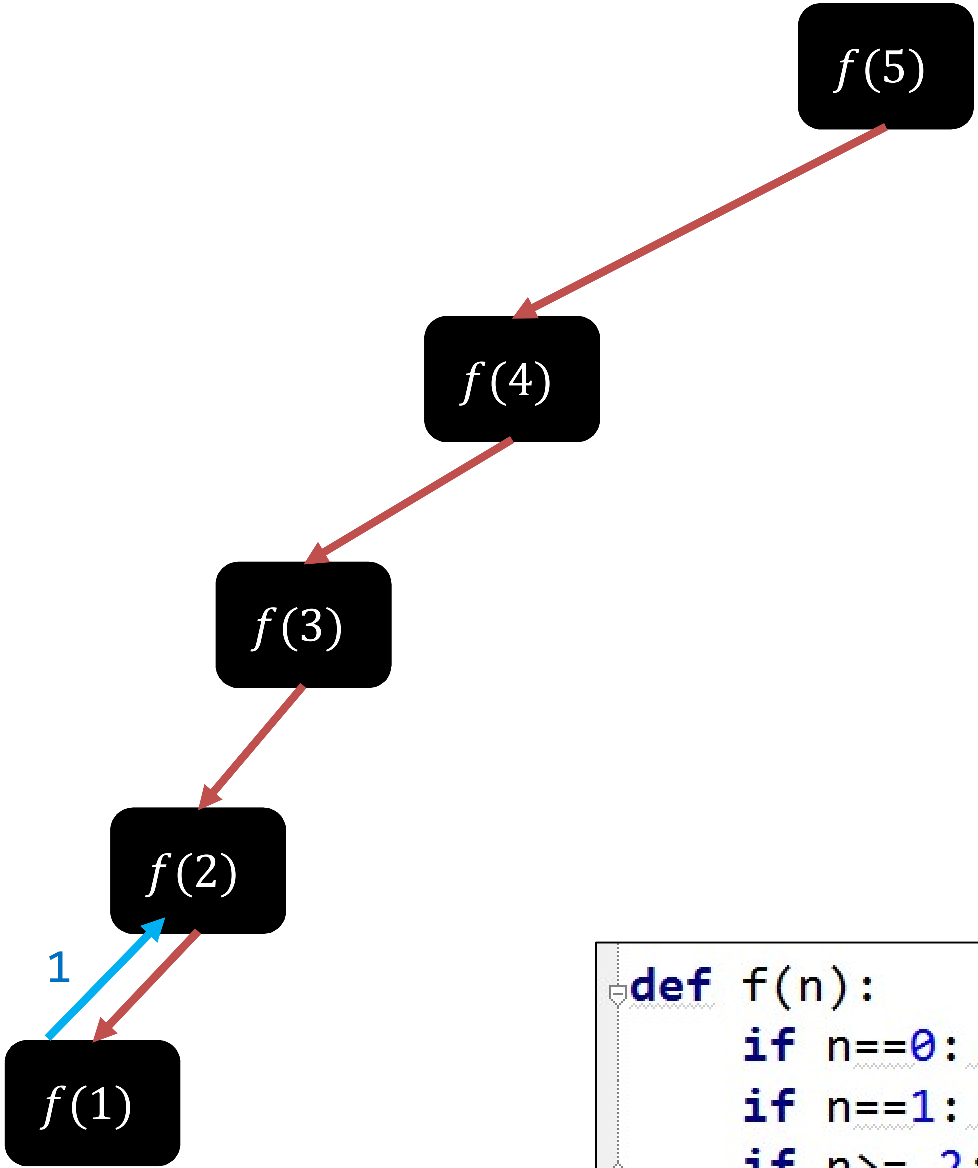
- An iterative algorithm for the Fibonacci numbers has running time of $O(n)$.
- But using recursion, each call $f(n)$ leads to another **two** calls: $f(n-1)$ and $f(n-2)$.

Example: Fibonacci Sequence



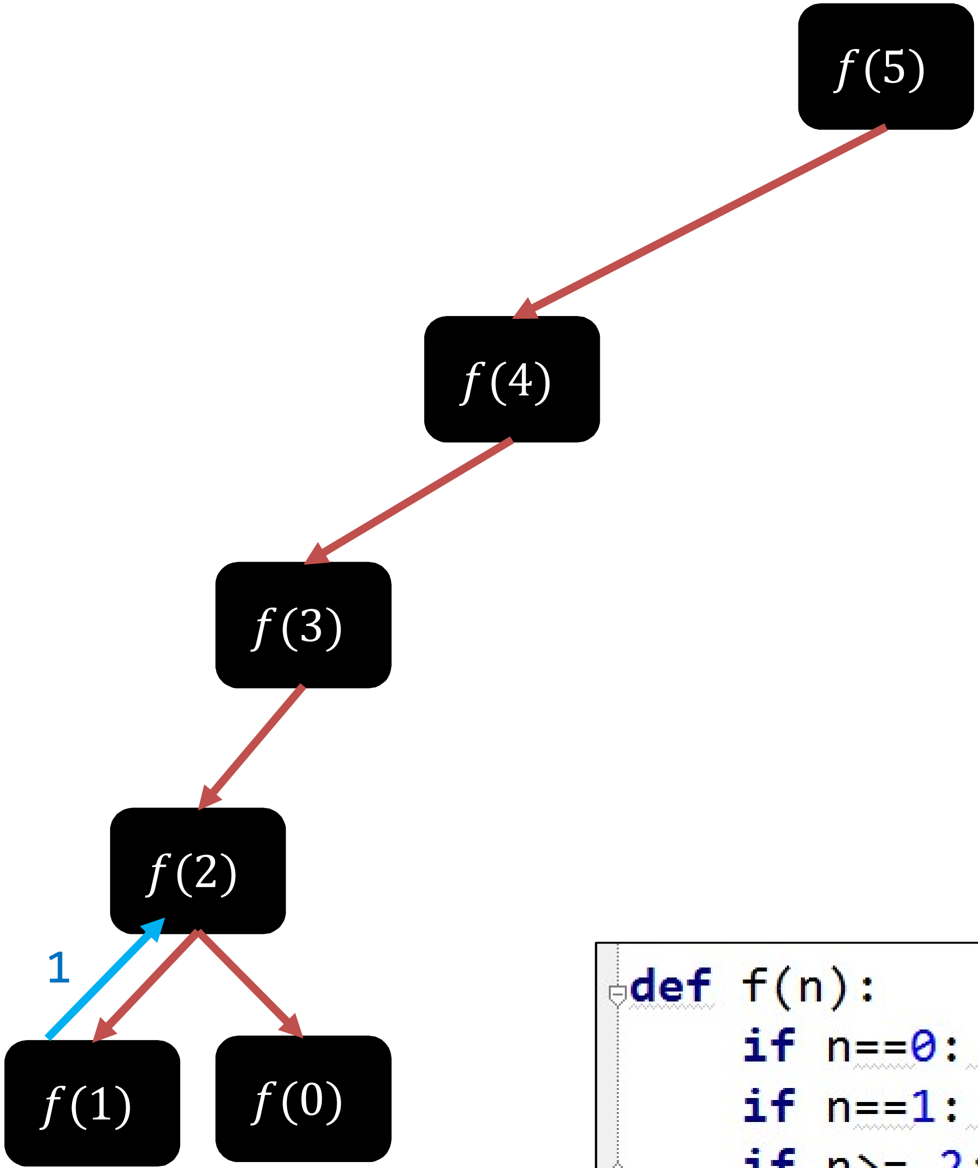

```
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

Example: Fibonacci Sequence



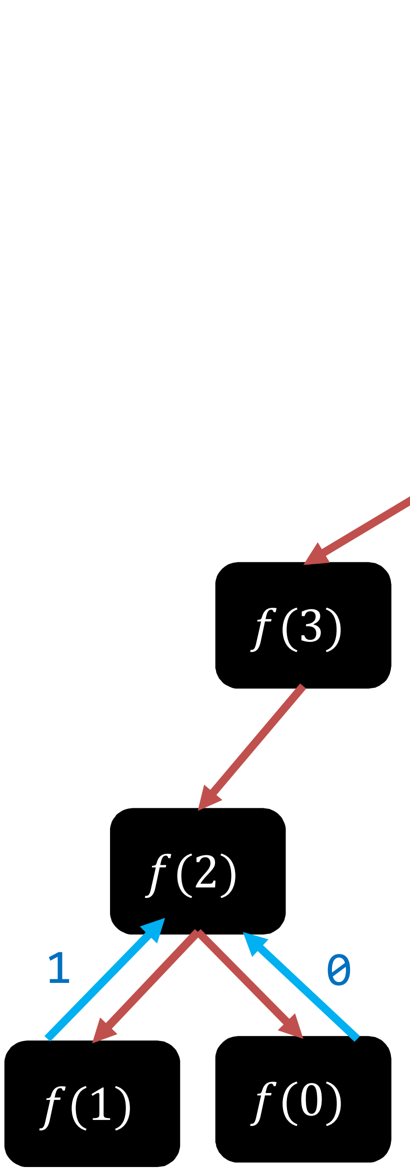

```
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

Example: Fibonacci Sequence



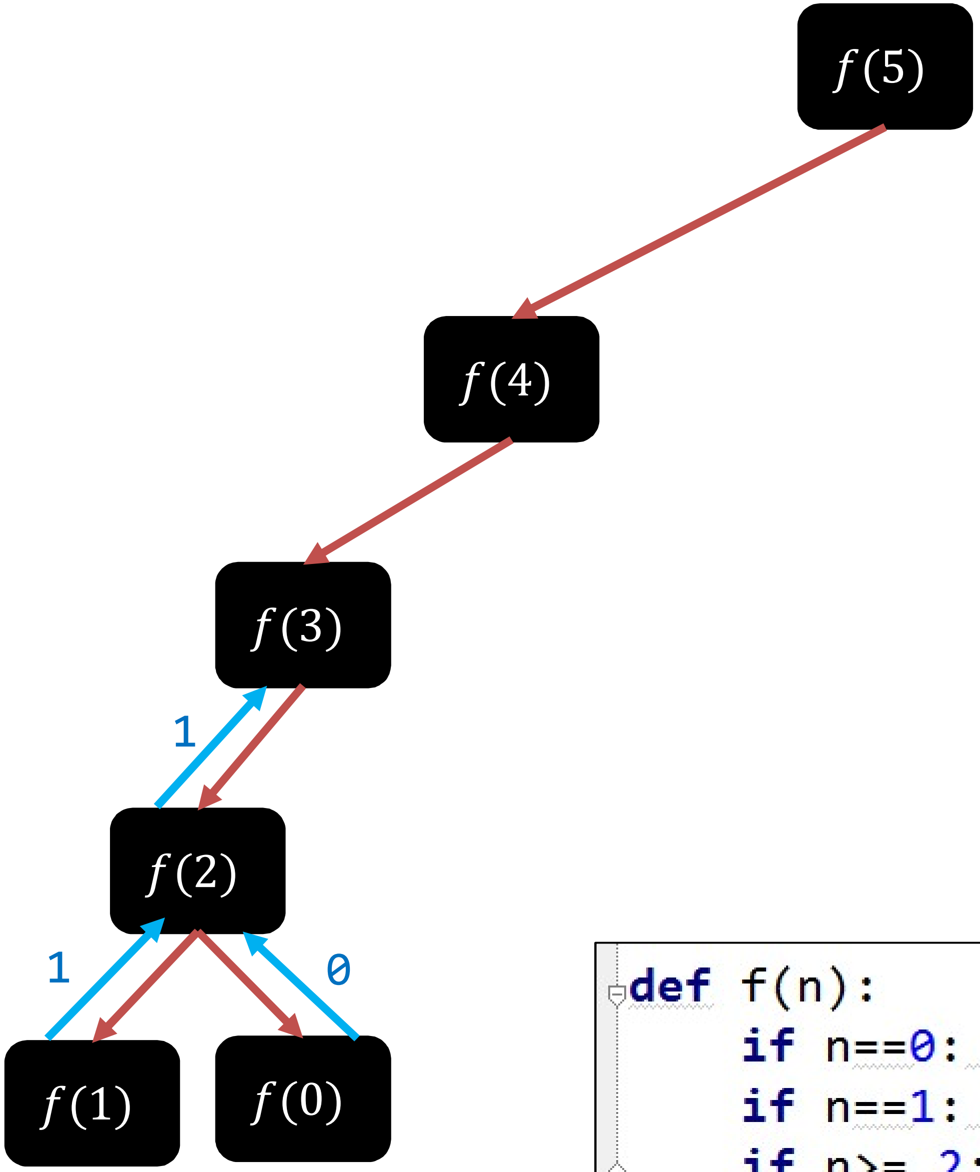

```
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

Example: Fibonacci Sequence



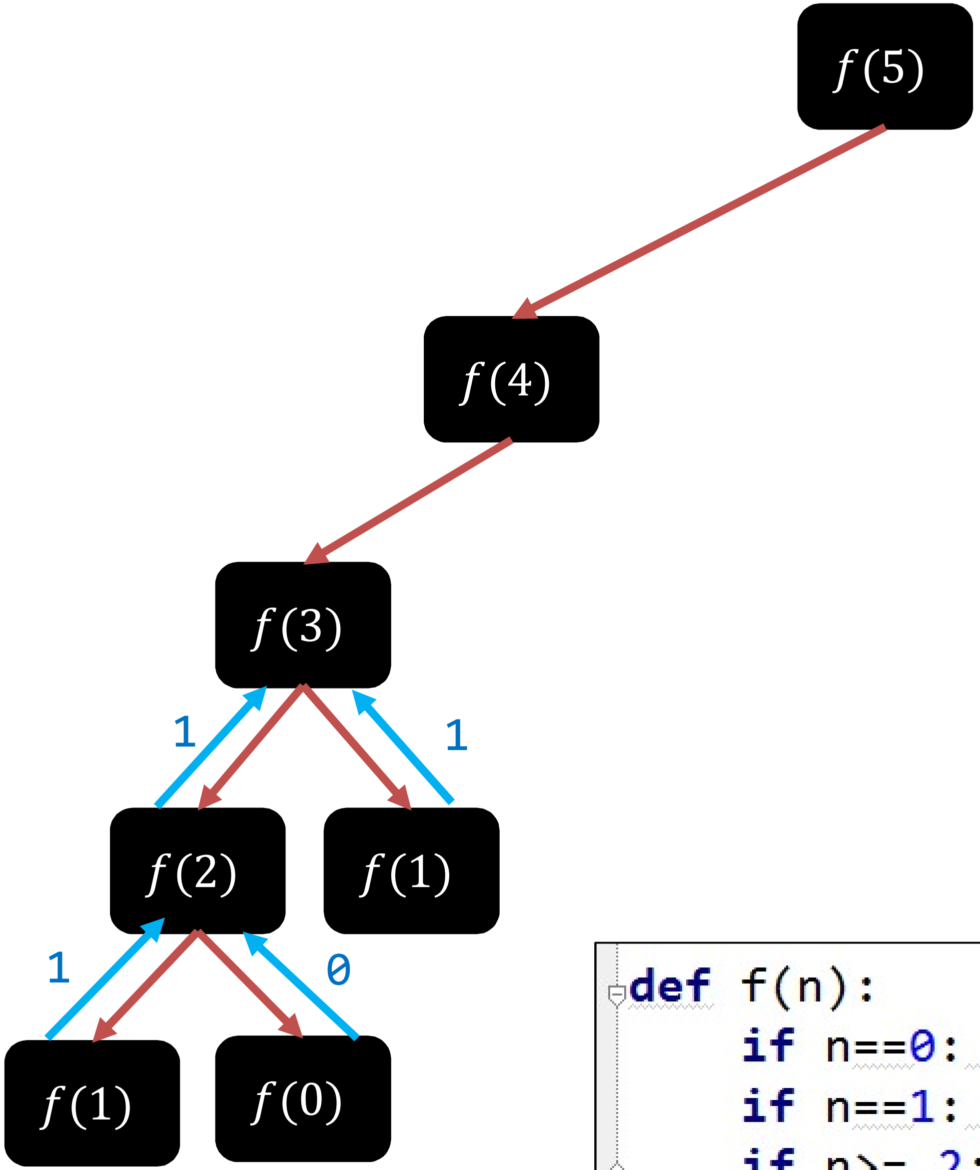

```python
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

Example: Fibonacci Sequence



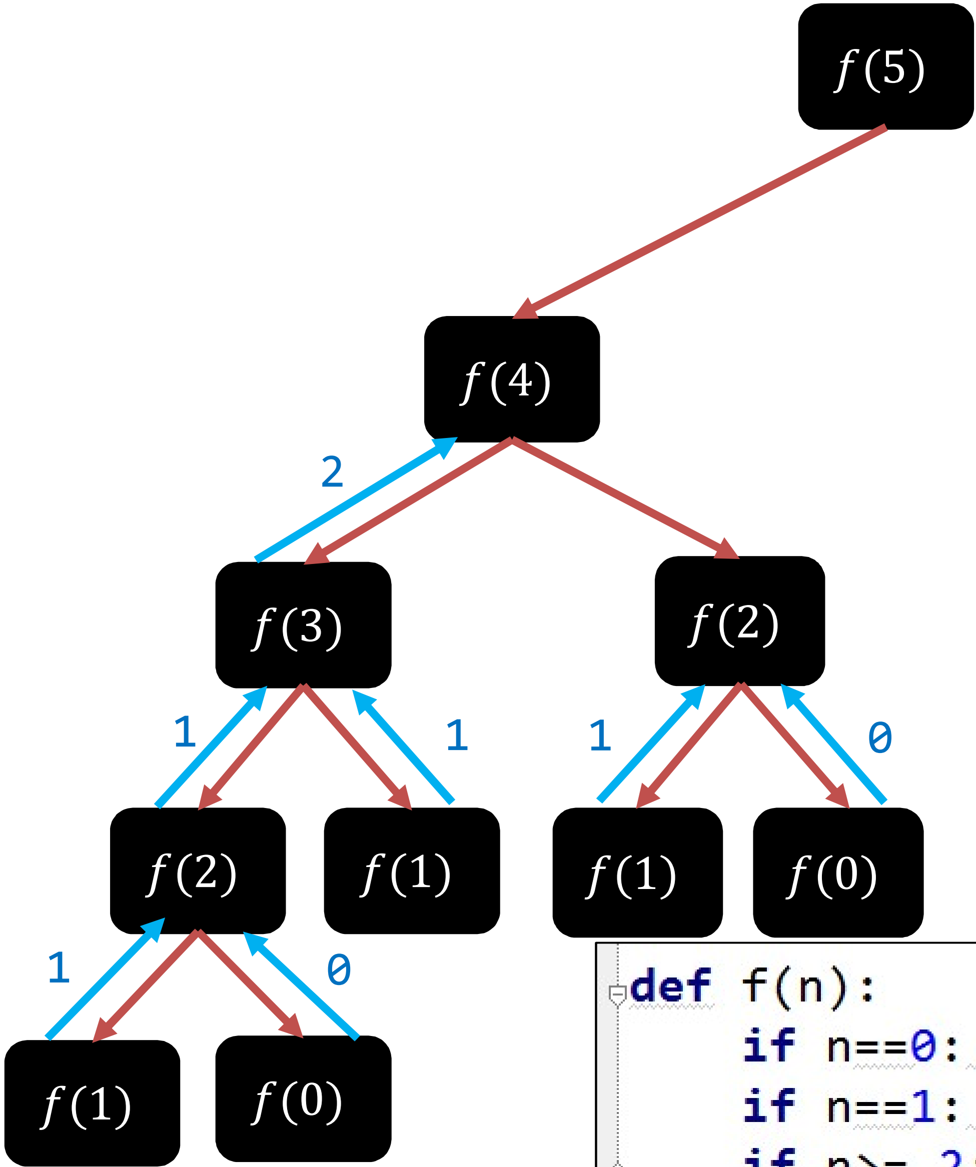

```
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

Example: Fibonacci Sequence



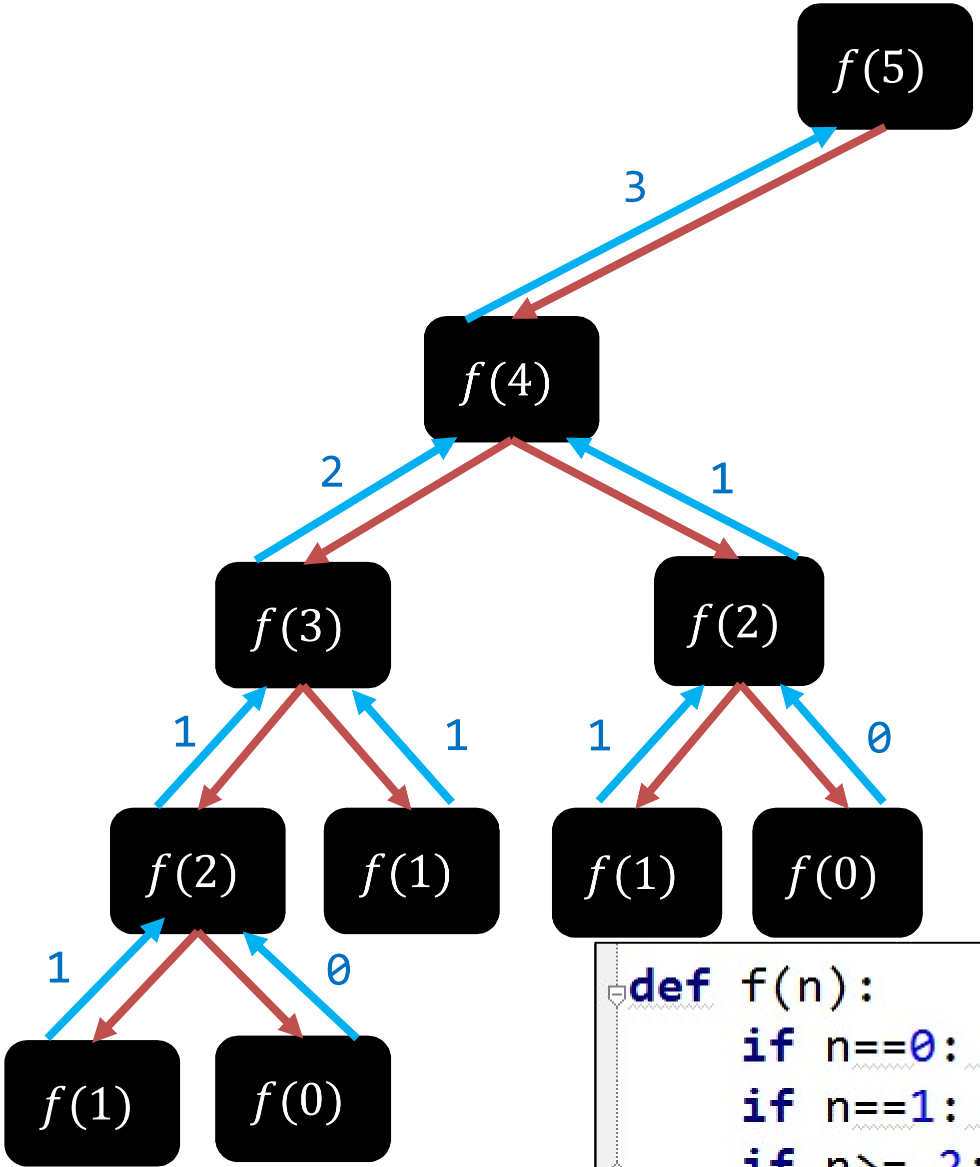

```
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

Example: Fibonacci Sequence



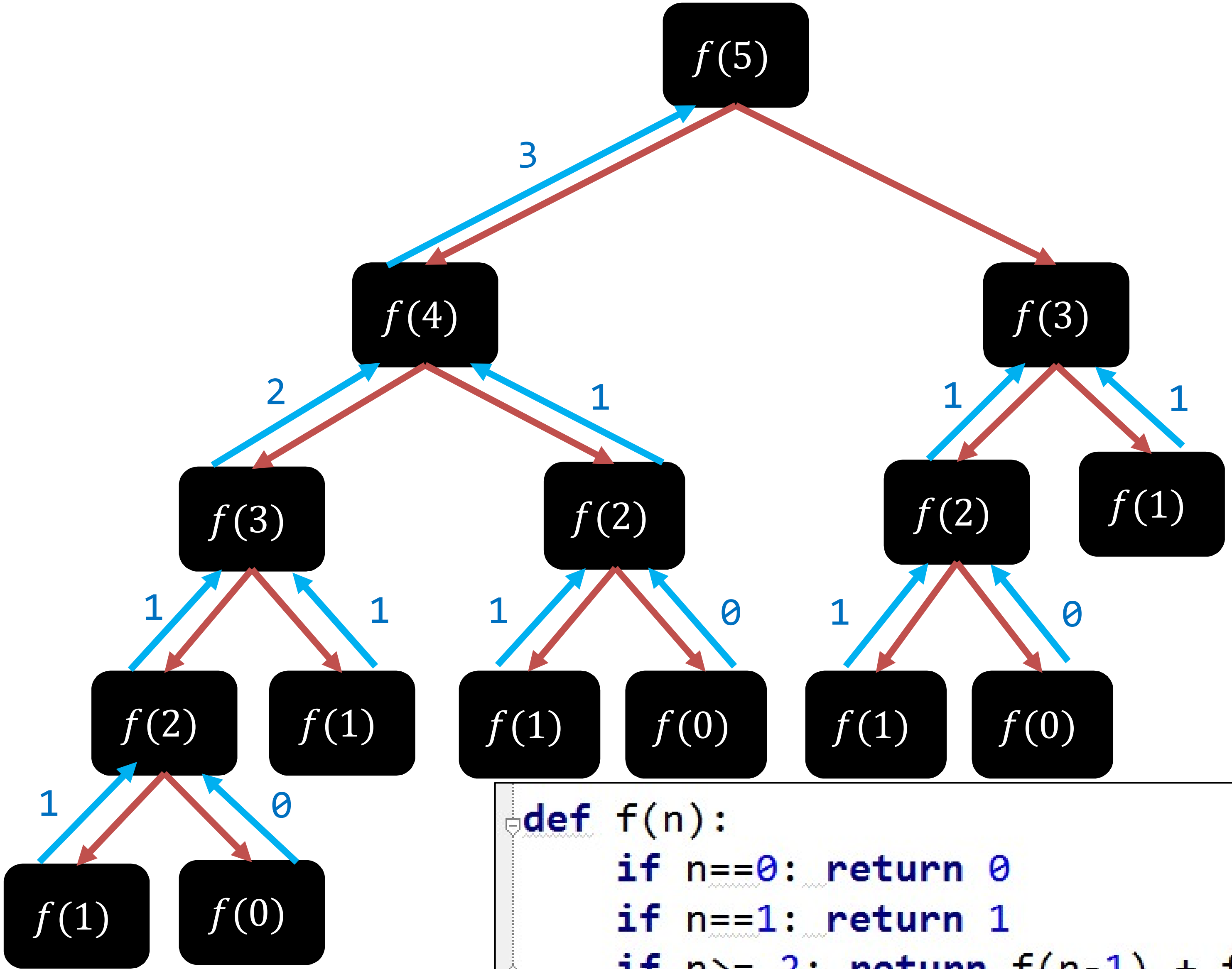

```python
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

Example: Fibonacci Sequence



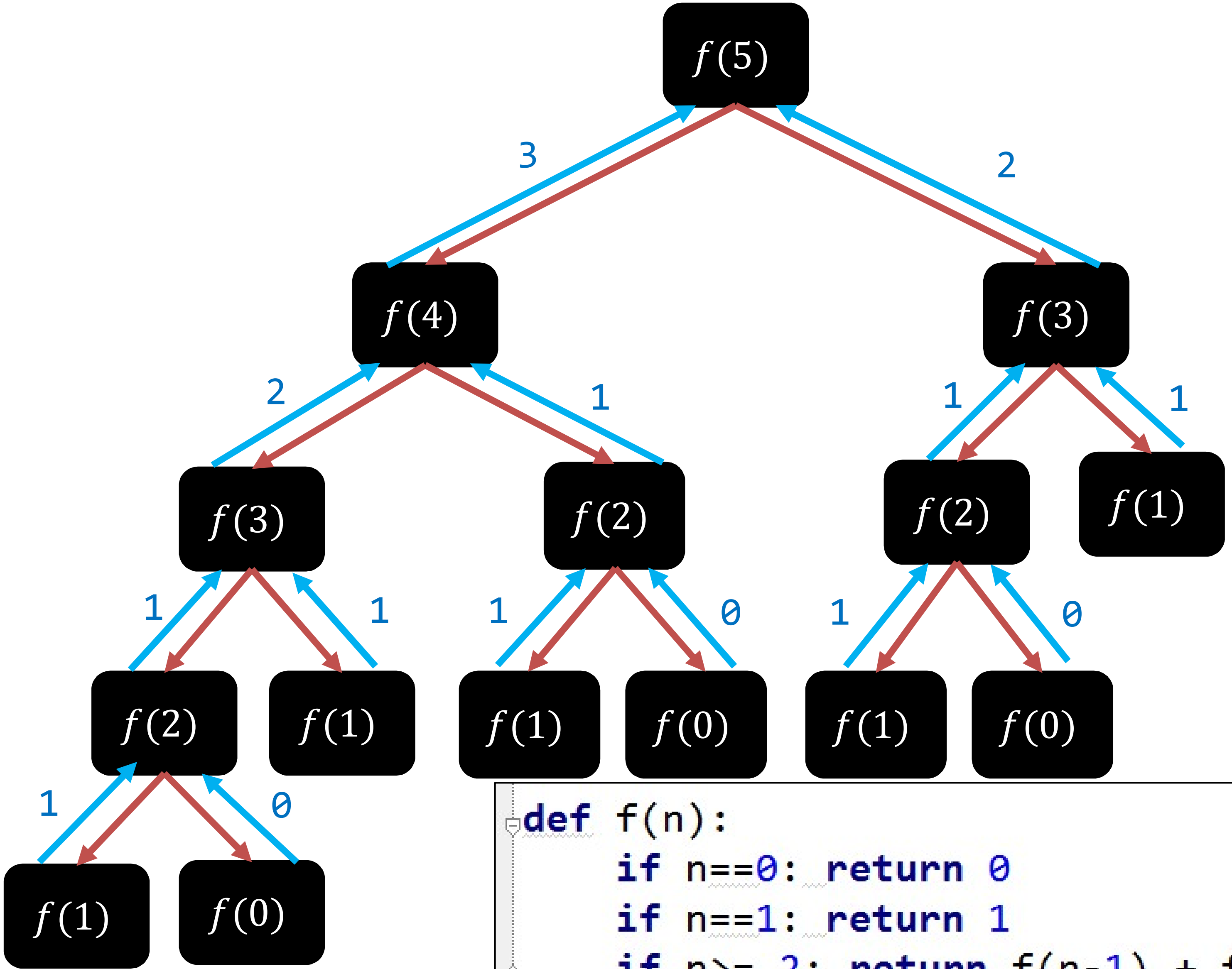

```python
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

Example: Fibonacci Sequence




```python
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

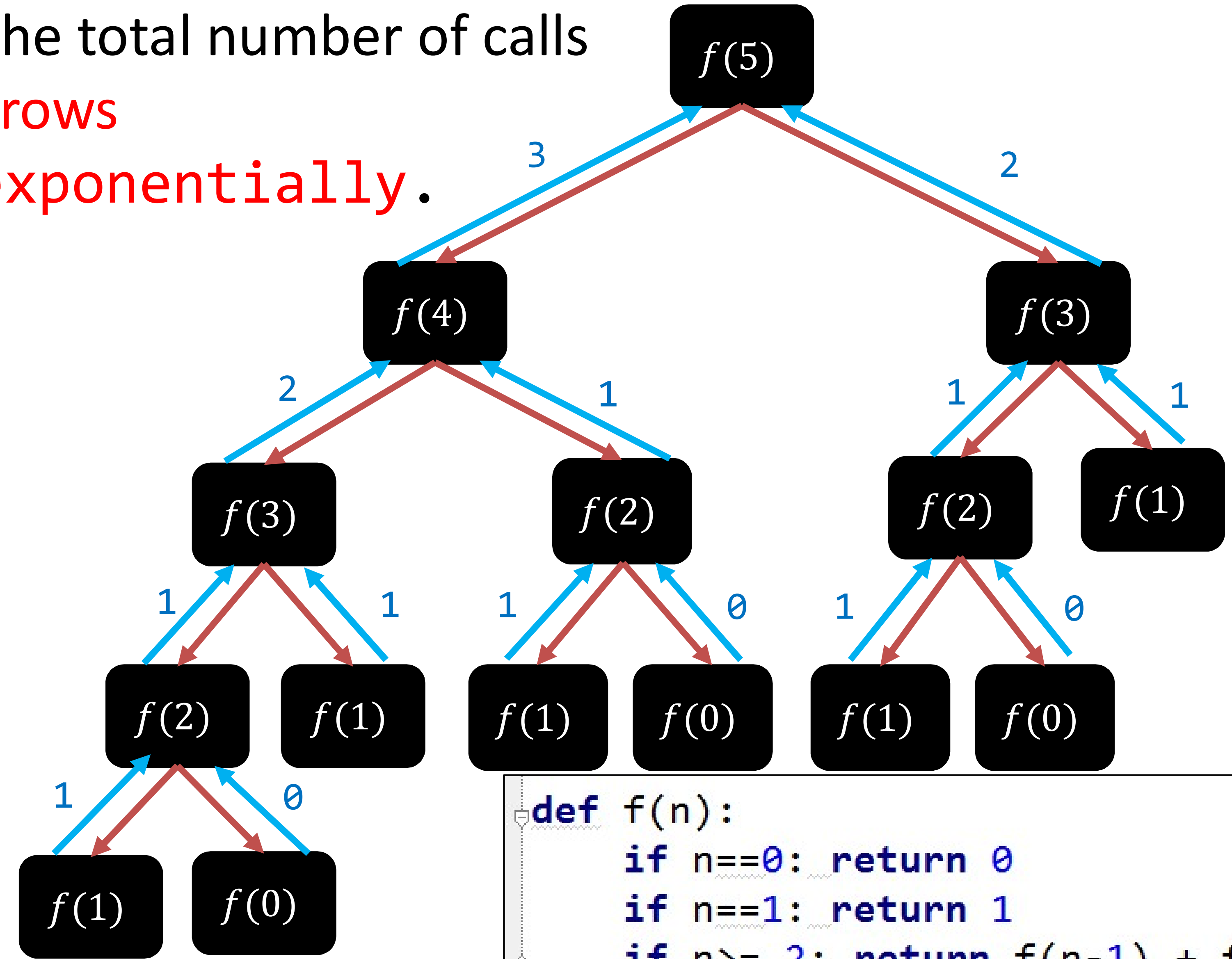
Example: Fibonacci Sequence




```python
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

Example: Fibonacci Sequence

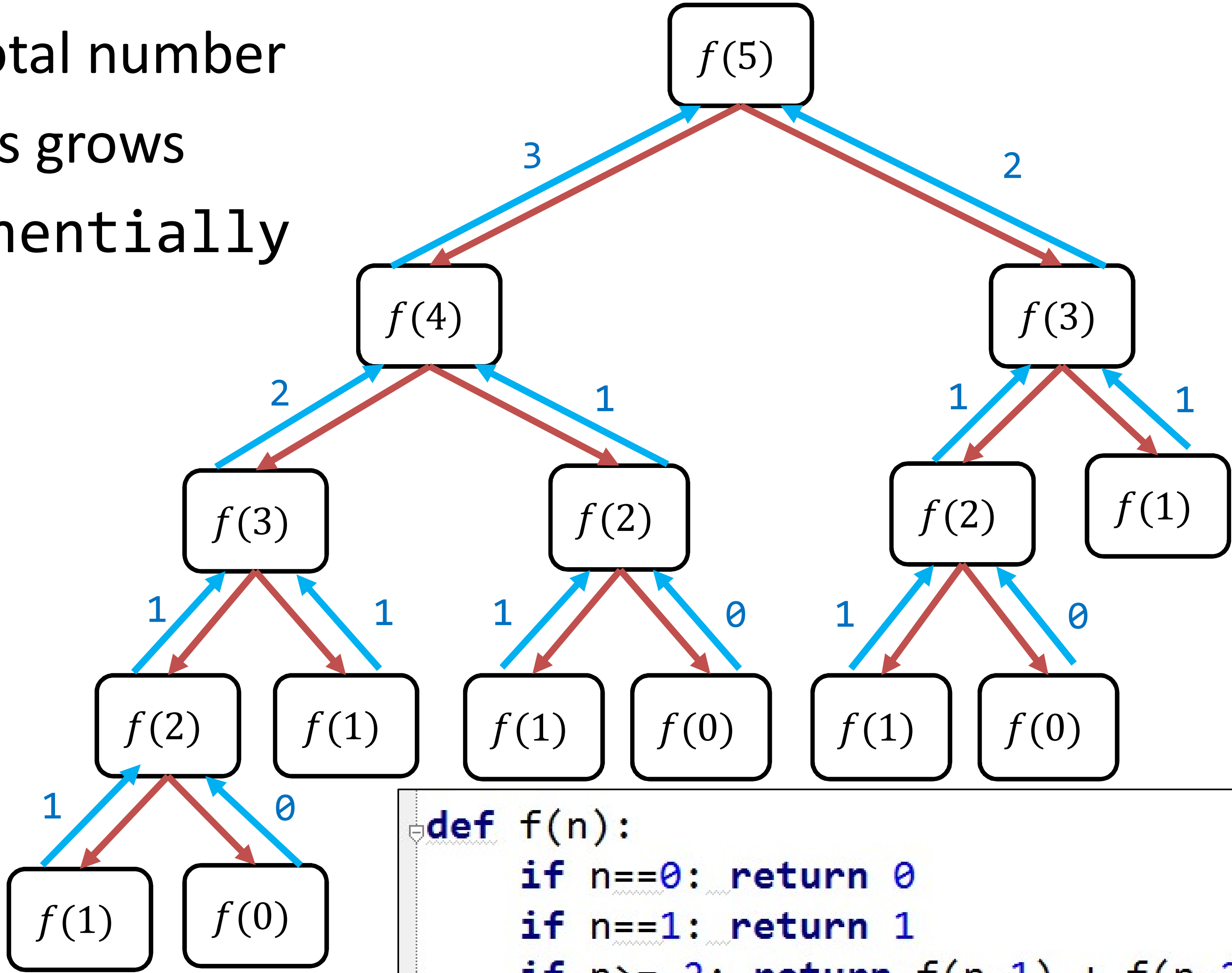
The total number of calls grows exponentially.




```
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

Example: Fibonacci Sequence

The total number of calls grows exponentially




```python
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

Example: Fibonacci Sequence

Running time of recursive Fibonacci algorithm

```
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

$$T(n) = T(n-1) + T(n-2)$$

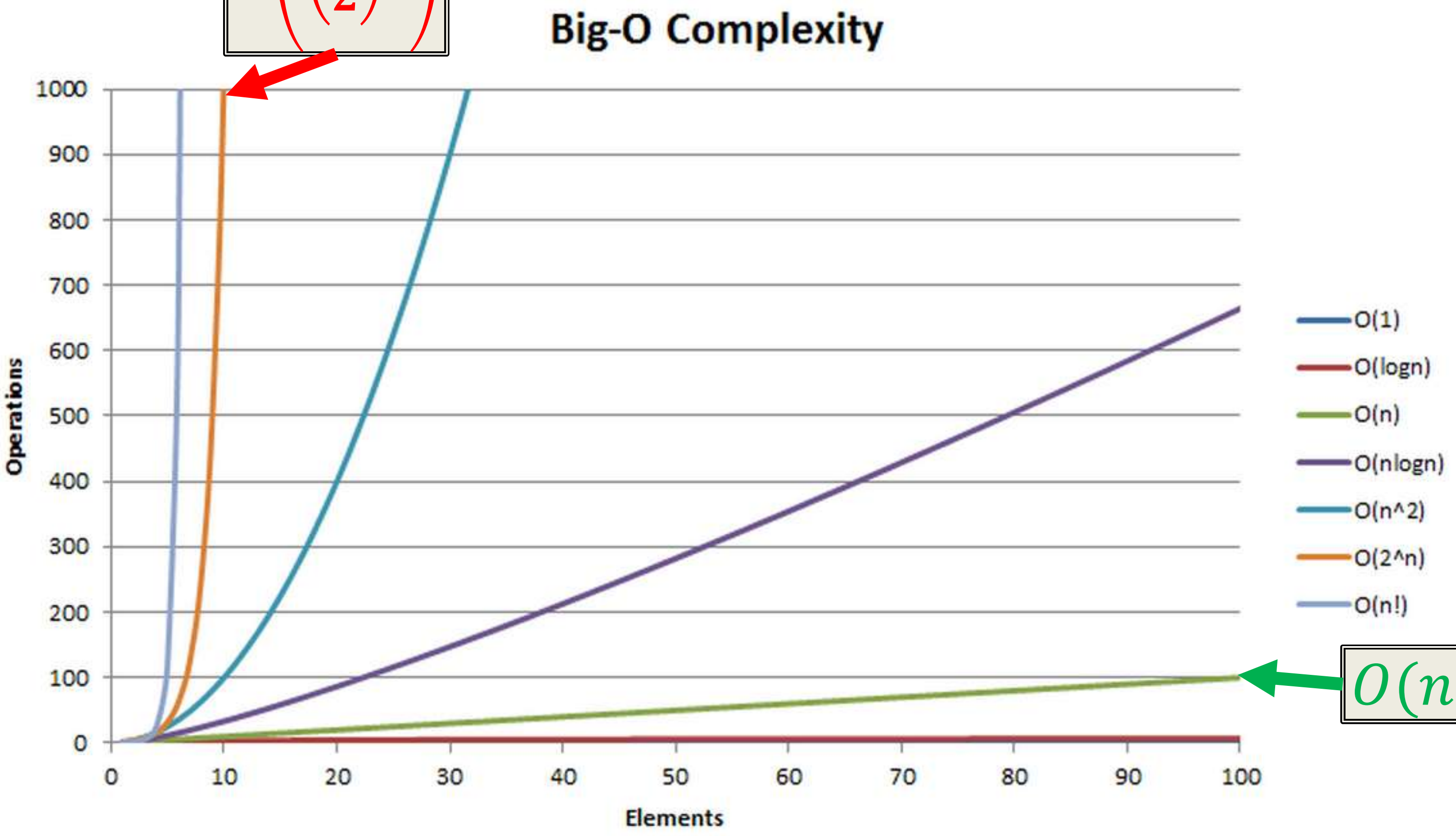
$$T(0) = 1, T(1) = 1$$

It can be shown that $T(n) = \Omega\left(\left(\frac{3}{2}\right)^n\right)$.

Compute Fibonacci: Recursive vs. Non-recursive

SINGAPORE INSTITUTE OF TECHNOLOGY

$$O\left(\left(\frac{3}{2}\right)^n\right)$$



$O(n)$

Optimization & Greedy Algorithms

- An optimization problem means to find *best* solution, not just *a* solution.
- A "greedy algorithm" sometimes works well for optimization problems.
- A greedy algorithm works in phases. At each phase:
  - take the best you can get right now, without regard for future consequences.
  - hope that choosing a *local* optimum at each step will end up at a *global* optimum.

Greedy = Optimal?

- Greedy algorithms
  do not always yield optimal solutions
  …although they do for many problems.
- Examples of Greedy Algorithms:
  – Dijkstra’s Shortest Path Algorithm.
  – Kruskal’s Minimum Spanning Tree Algorithm.
  – Prim’s Minimum Spanning Tree Algorithm.

Greedy Algorithm to Count Money

Suppose we want to gather an amount of money, using the fewest possible bills and coins.

- A greedy algorithm to do it:
  At each step, take the largest possible bill or coin that does not overshoot.
  eg. to form $6.39, we choose (for US$):
  - a $5 bill
  - a $1 bill, = $6
  - a 25¢ coin, = $6.25
  - a 10¢ coin, = $6.35
  - four 1¢ coins, = $6.39 ; total 8 pcs (bills & coins)
- For US money, the greedy algorithm always gives the optimal solution.

Failure of Greedy Algorithm

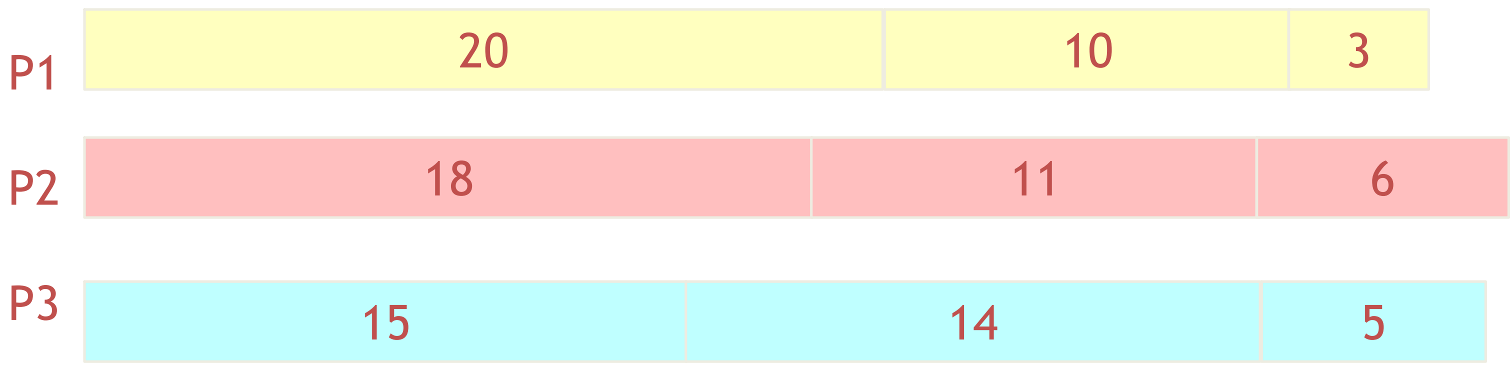
Suppose some foreign currency uses \$1, \$7, \$10 coins.

- A greedy algorithm to form \$15:
  one \$10 + five \$1 coins = 6 coins.
- A better solution:
  two \$7 + one \$1 = 3 coins.
- The greedy algorithm gives a solution,
  but not an optimal solution.

Greedy Algorithm for Scheduling Problem

Task: To execute nine jobs with these running times
3, 5, 6, 10, 11, 14, 15, 18, 20 minutes.
Resources: 3 processors to run the jobs.

- Approach 1: Do longest jobs first,
  on whatever processor is available.



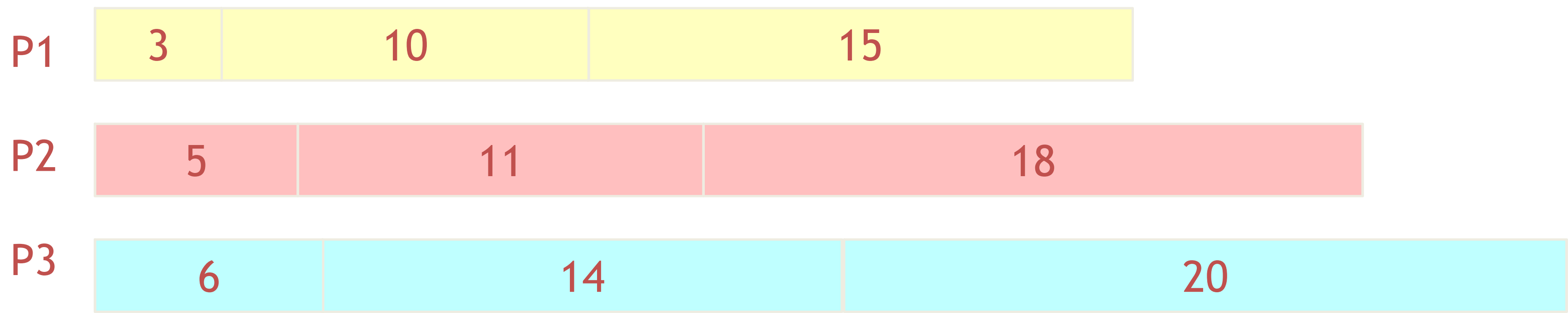
Time to completion: 18 + 11 + 6 = 35 minutes.

Is there a better solution?

Second Approach

- Approach 2: Do shortest jobs first.

(3, 5, 6, 10, 11, 14, 15, 18, 20 minutes)

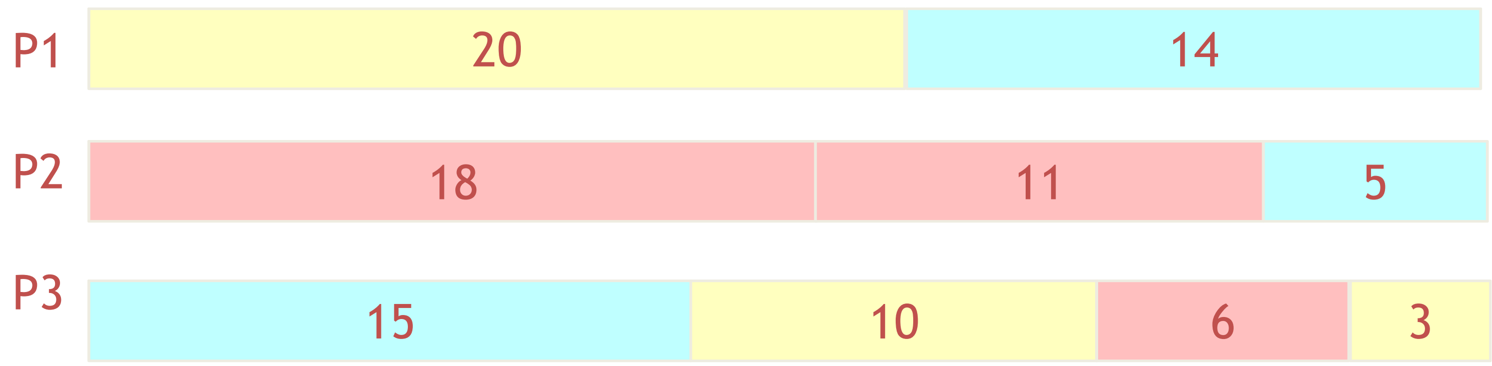


Not good; time needed is 6 + 14 + 20 = 40 minutes.

Note, however, that the greedy algorithm itself is fast; at each stage, just pick the minimum or maximum.

An Optimal Solution

- Better solutions do exist: (3, 5, 6, 10, 11, 14, 15, 18, 20 minutes)



- This solution is clearly optimal. (why?)
- Clearly, there are other optimal solutions. (why?)
- How do we find such a solution?
  - One way: Try all possible assignments of jobs to processors.
  - Unfortunately, this approach can take exponential time.