SINGAPORE INSTITUTE OF TECHNOLOGY

Recursion

Recurrence relations: Fibonacci sequence

Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34…

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

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

$$f_n = (f_{n-1} + f_{n-2}) \text{ for } n \geq 2$$
$$f_0 = 0$$
$$f_1 = 1$$

initial condition

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.

Example

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:

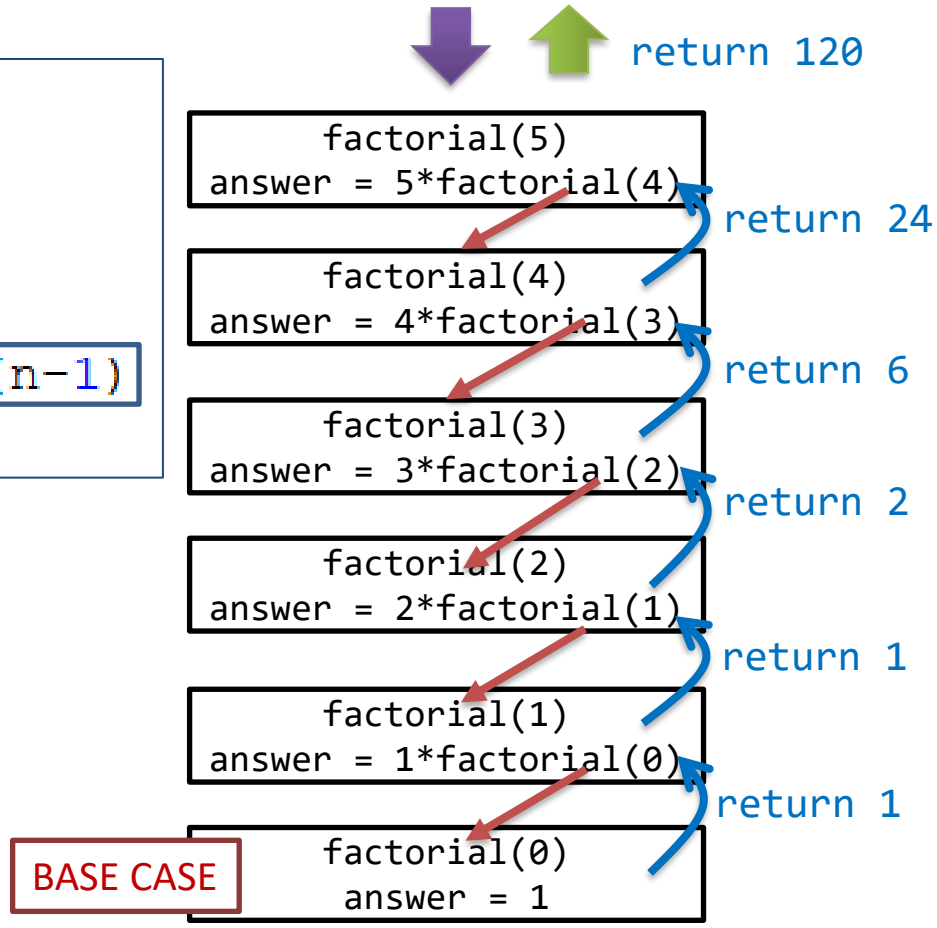
```
def factorial(n):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)
```

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!

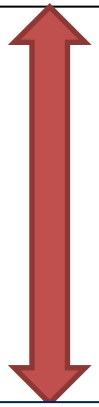
Using the method of `backward substitutions`

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

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:

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

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

```
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 vs. Iterative

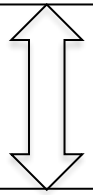
One should be careful with recursive algorithms because their conciseness, clarity and simplicity may hide their inefficiencies.

Example: Fibonacci Sequence

Recall that:

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

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

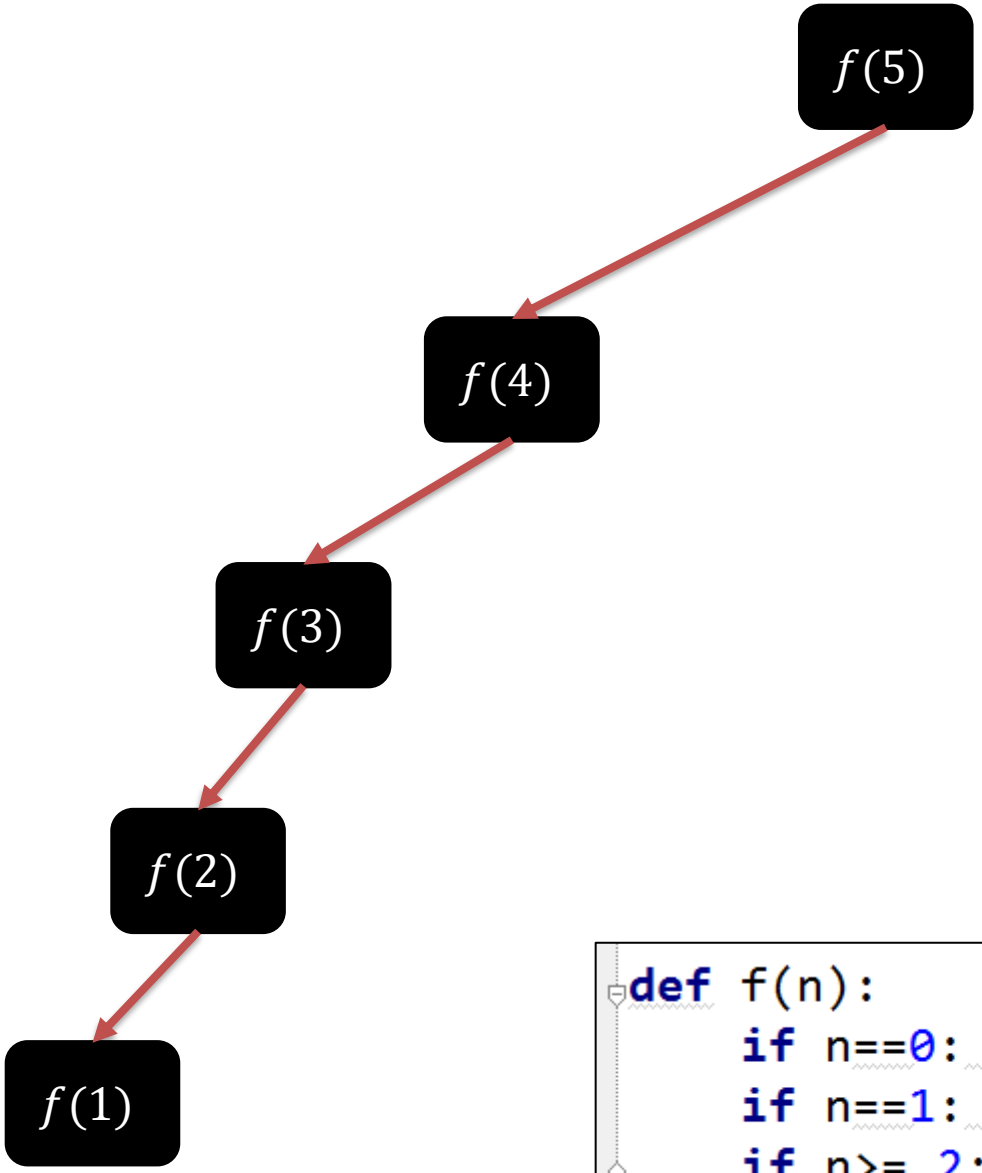


```
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

Example: Fibonacci Sequence

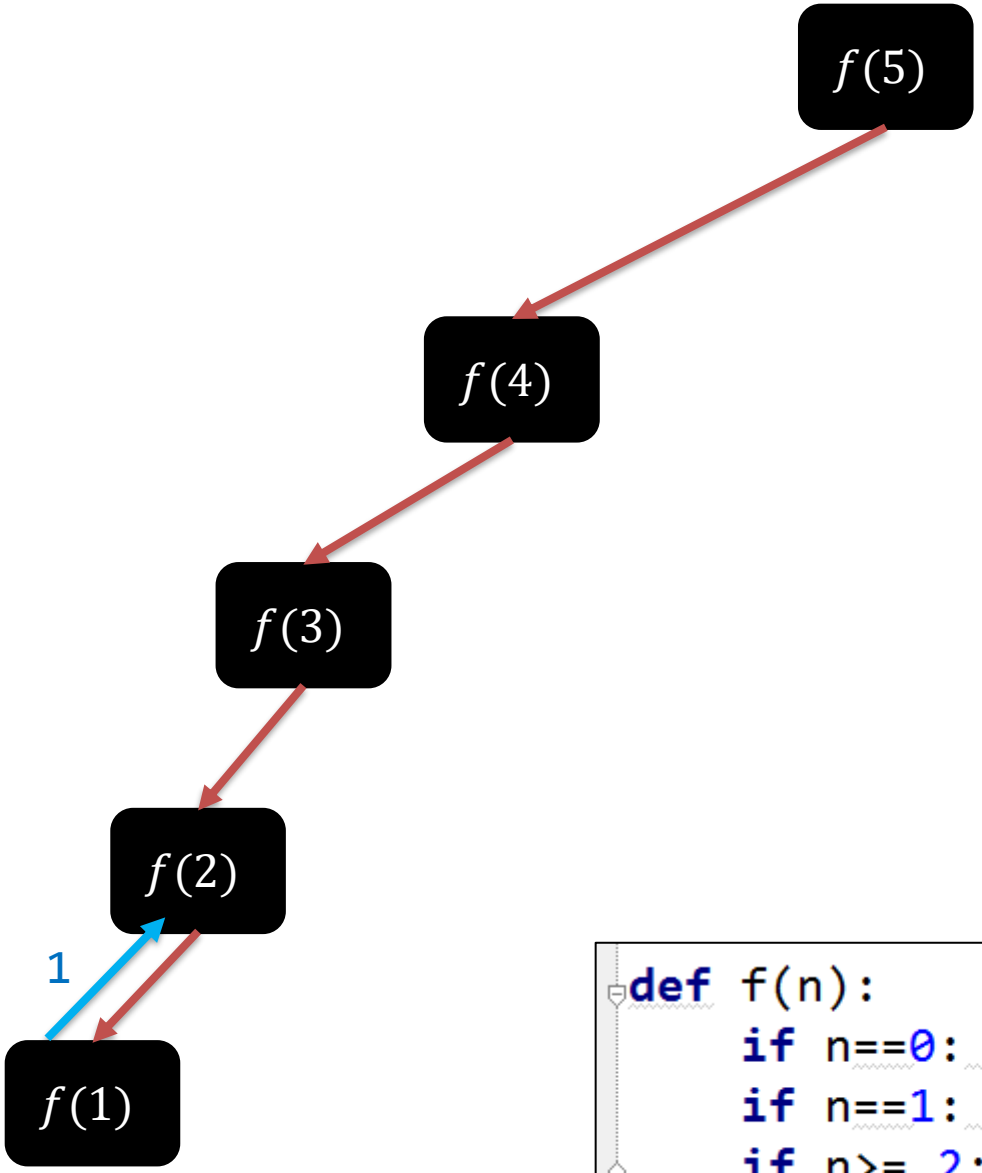
- An iterative algorithm for the Fibonacci numbers has running time of $O(n)$.
- But using recursion, each call $f(n)$ leads to another **two** calls: $f(n-1)$ and $f(n-2)$.

Example: Fibonacci Sequence



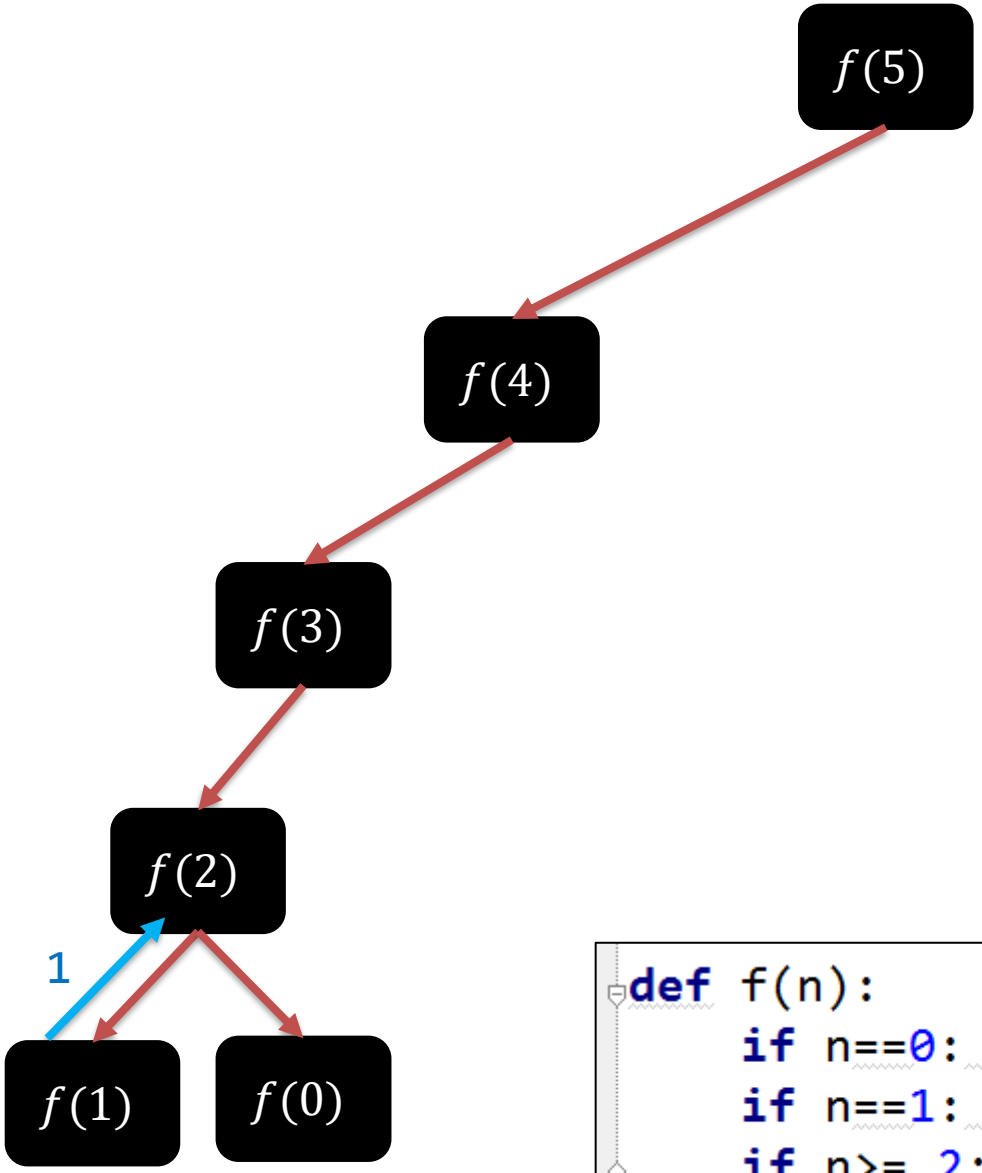

```
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

Example: Fibonacci Sequence



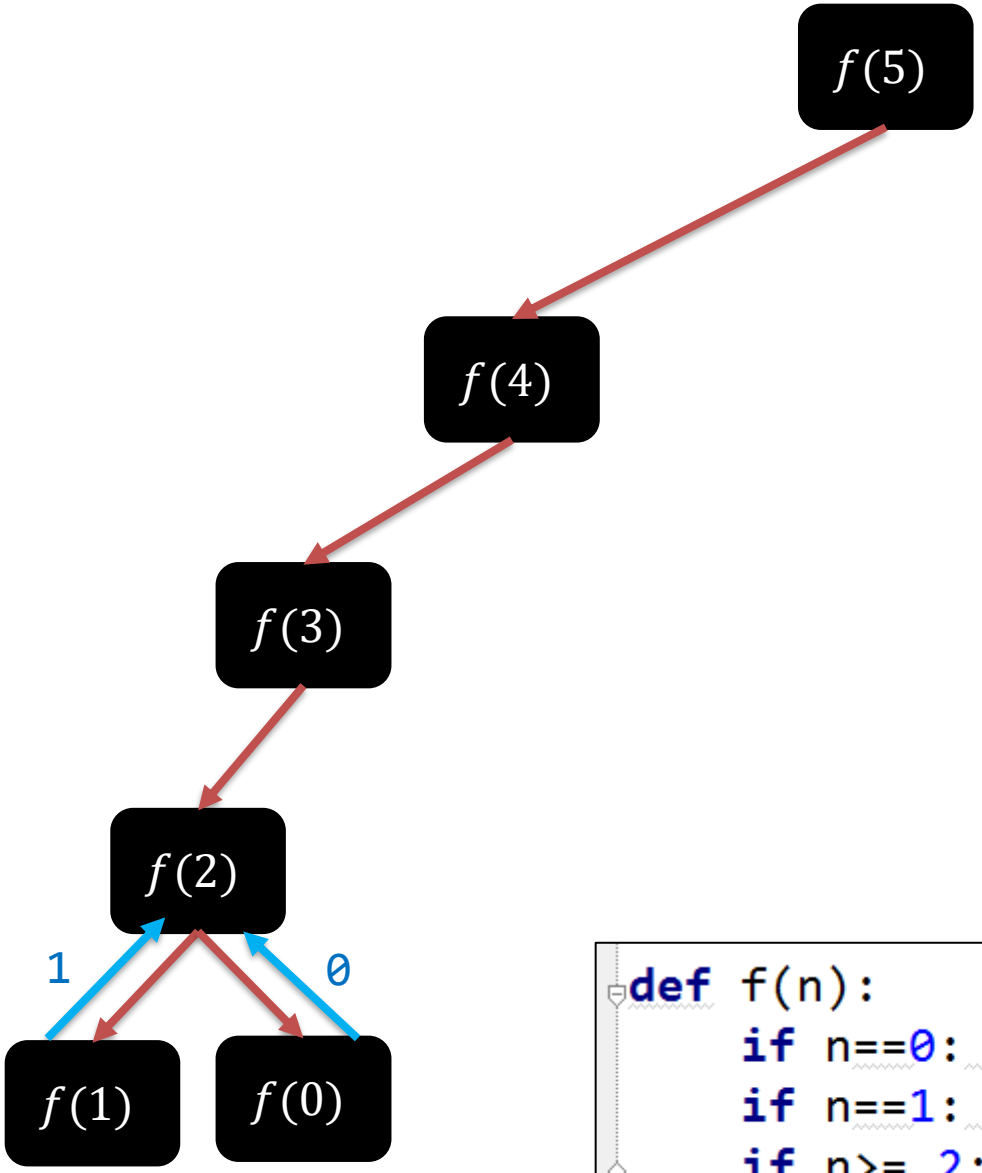

```
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

Example: Fibonacci Sequence



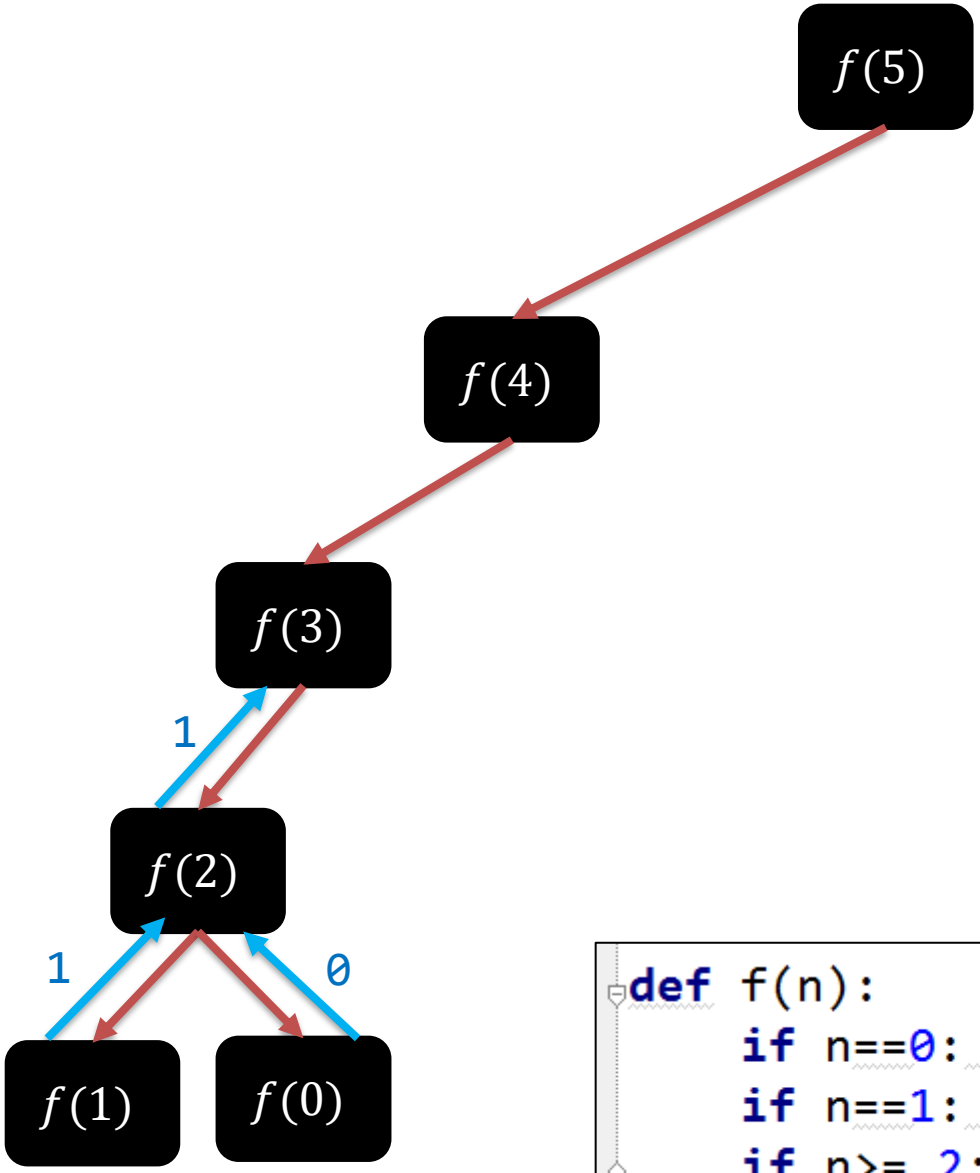

```
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

Example: Fibonacci Sequence



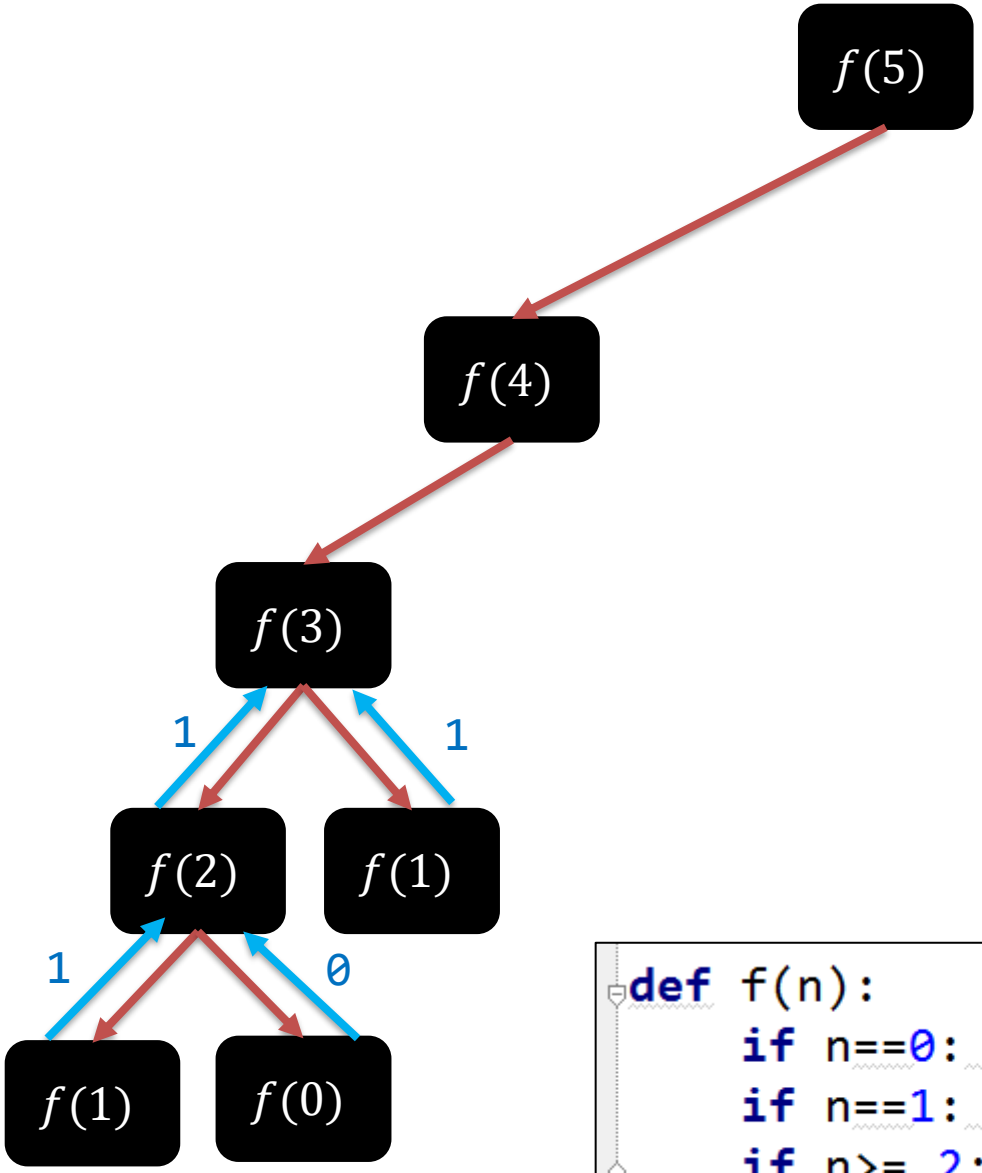

```
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

Example: Fibonacci Sequence



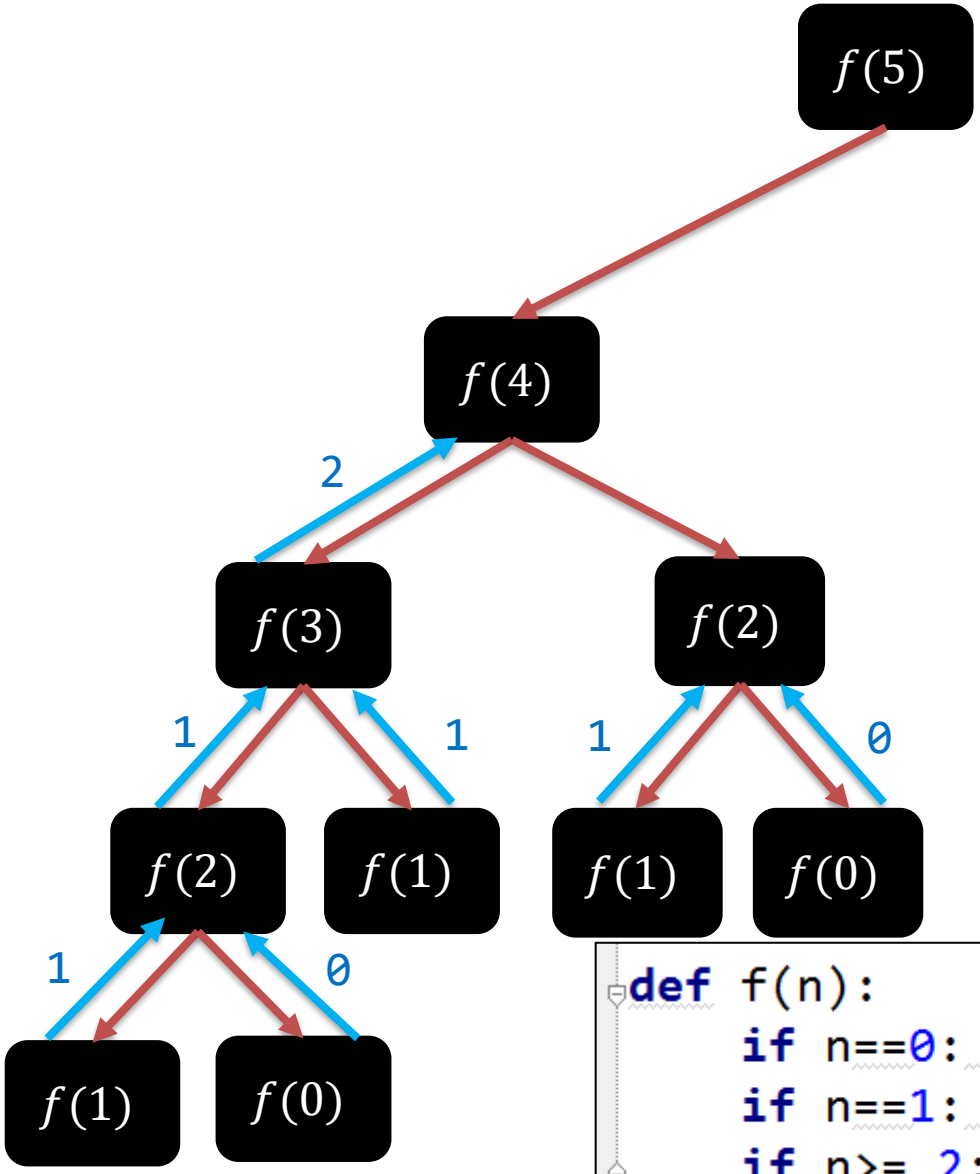

```
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

Example: Fibonacci Sequence



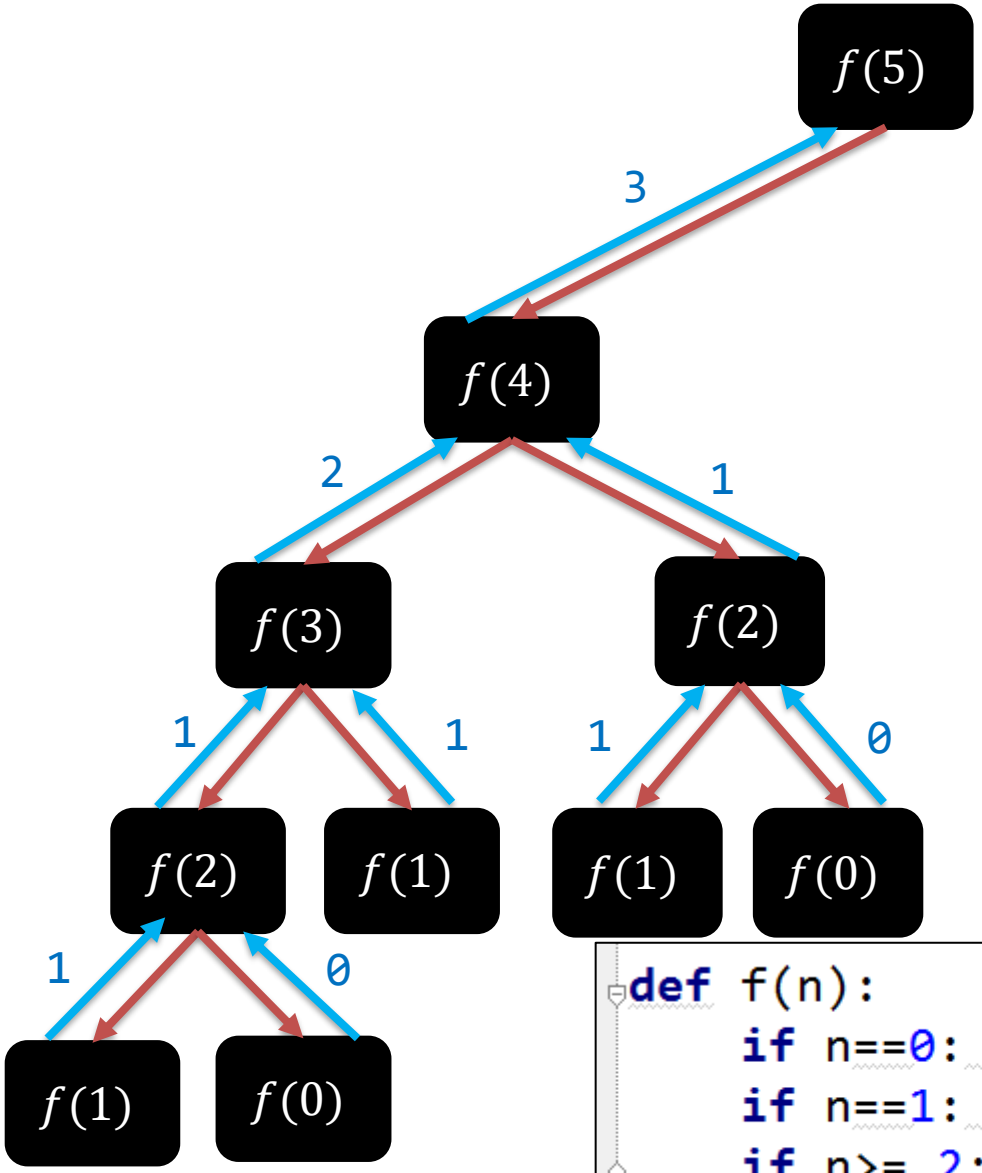

```
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

Example: Fibonacci Sequence



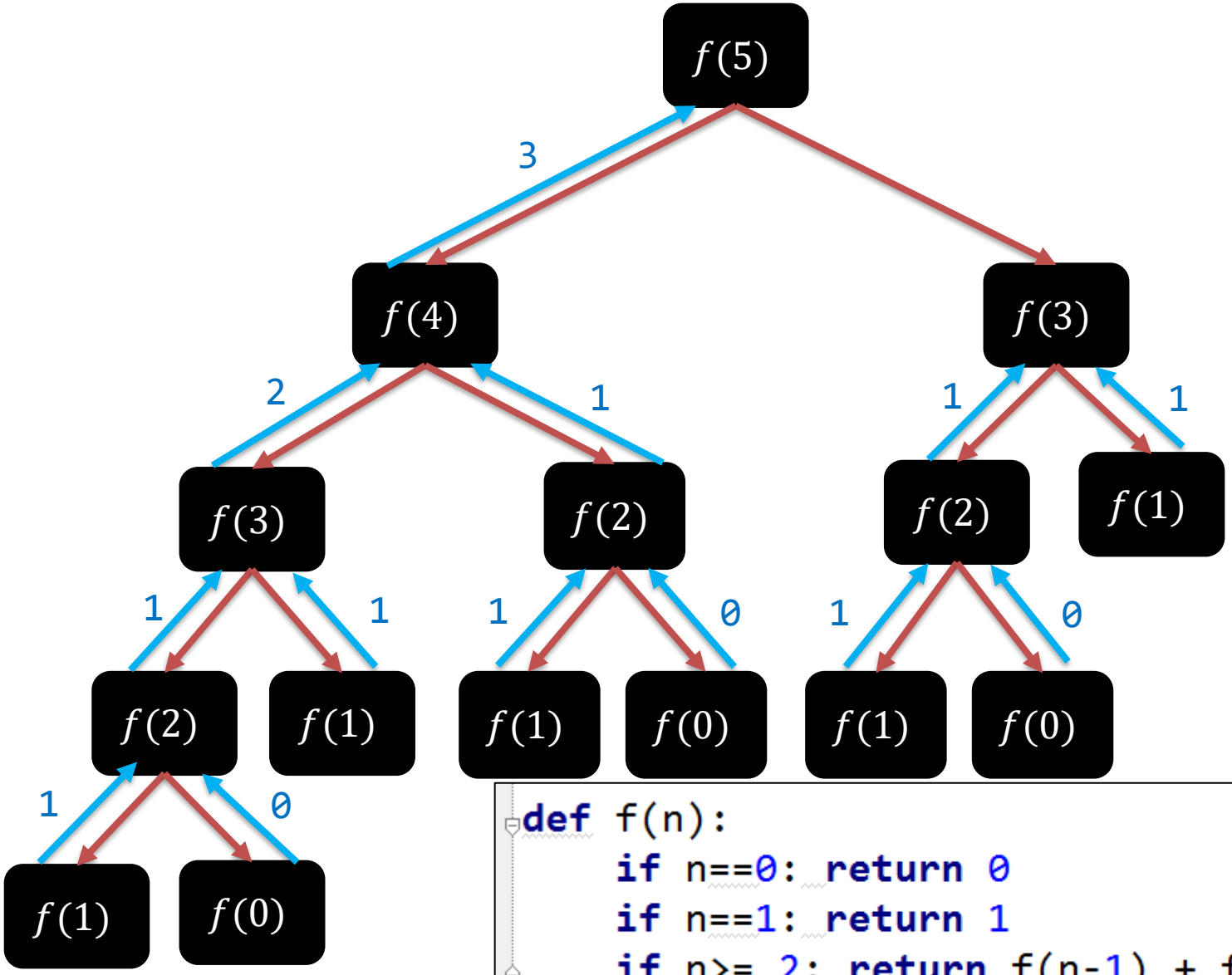

```
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

Example: Fibonacci Sequence



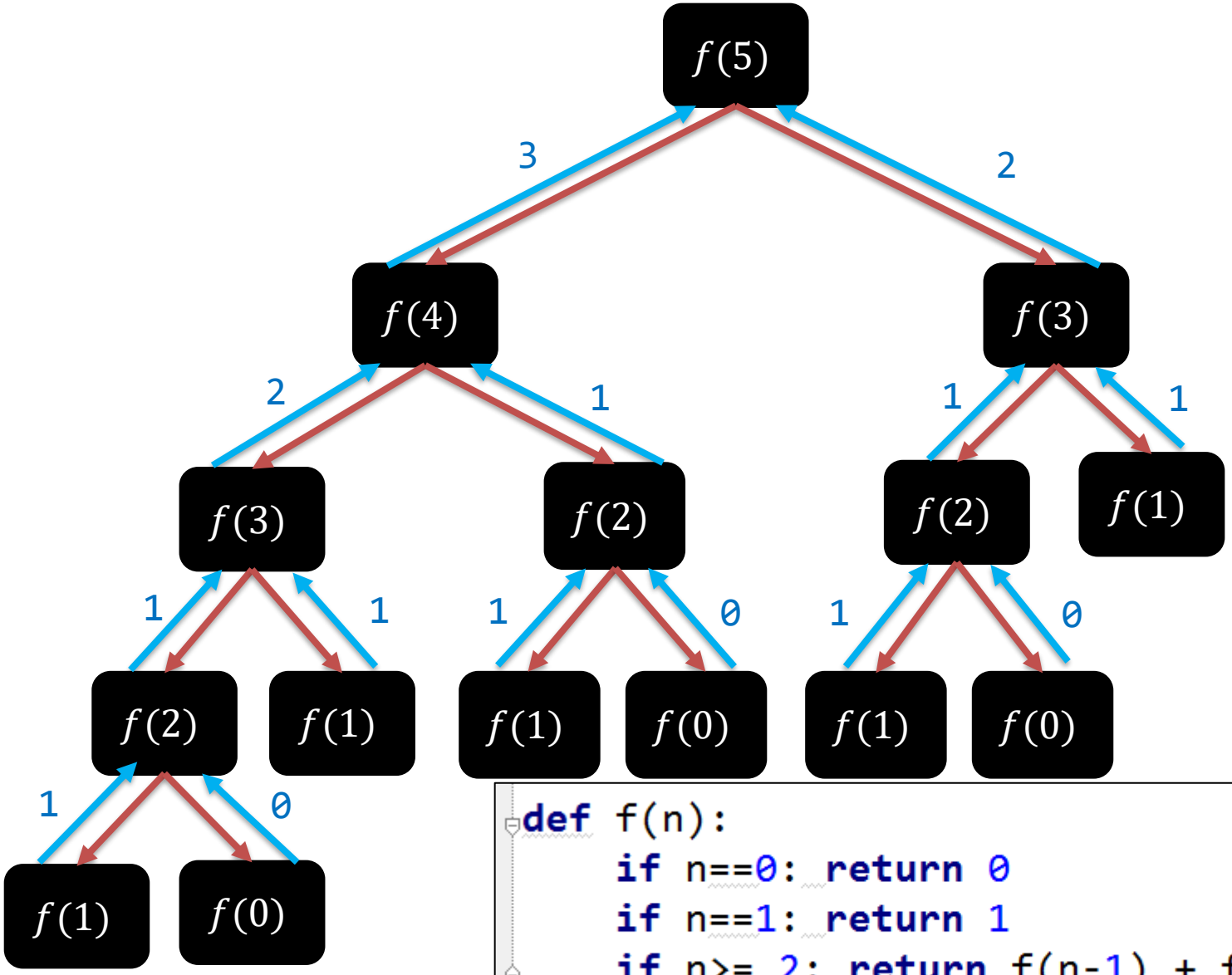

```
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

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)
```

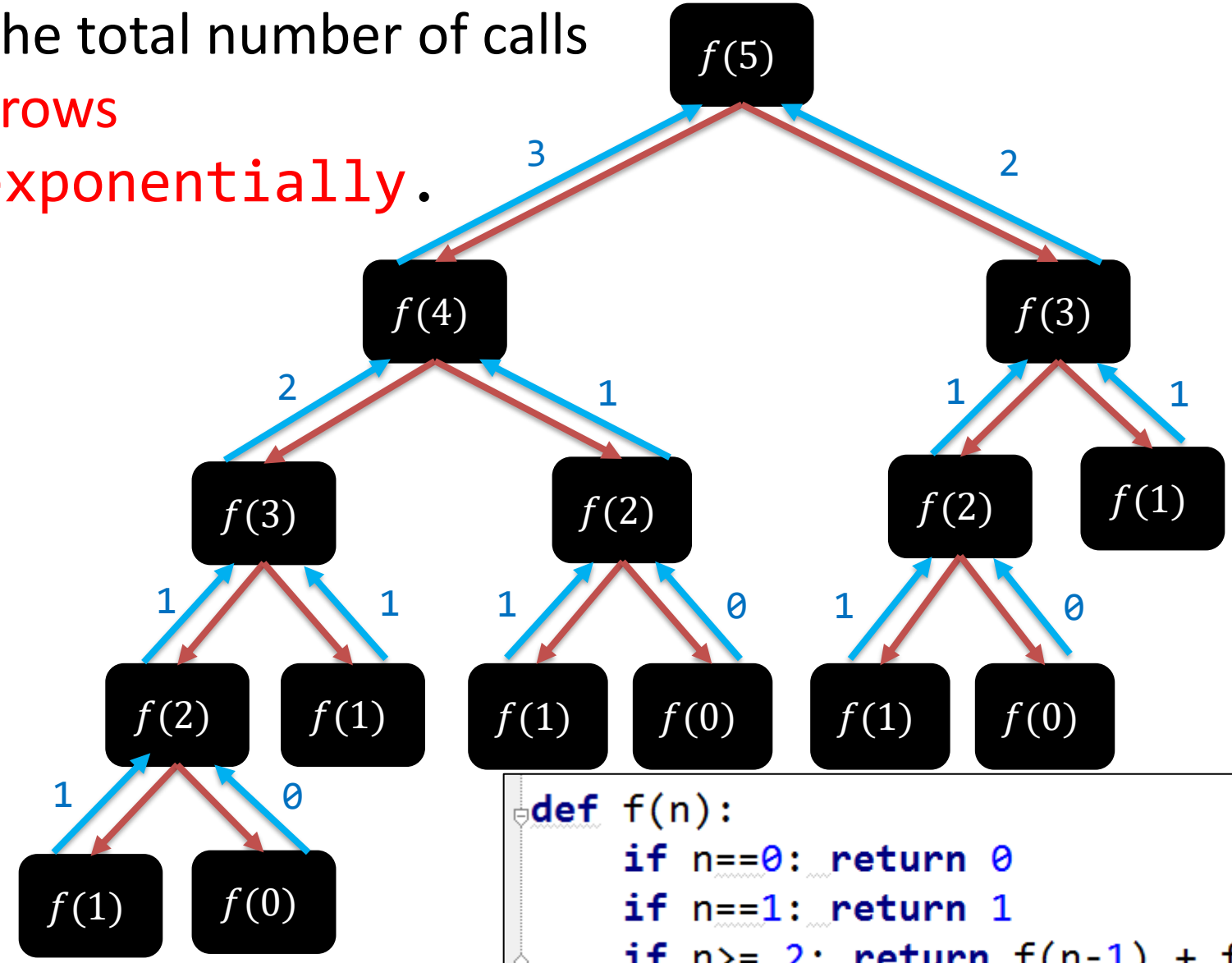
Example: Fibonacci Sequence




```
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

Running time of recursive Fibonacci algorithm

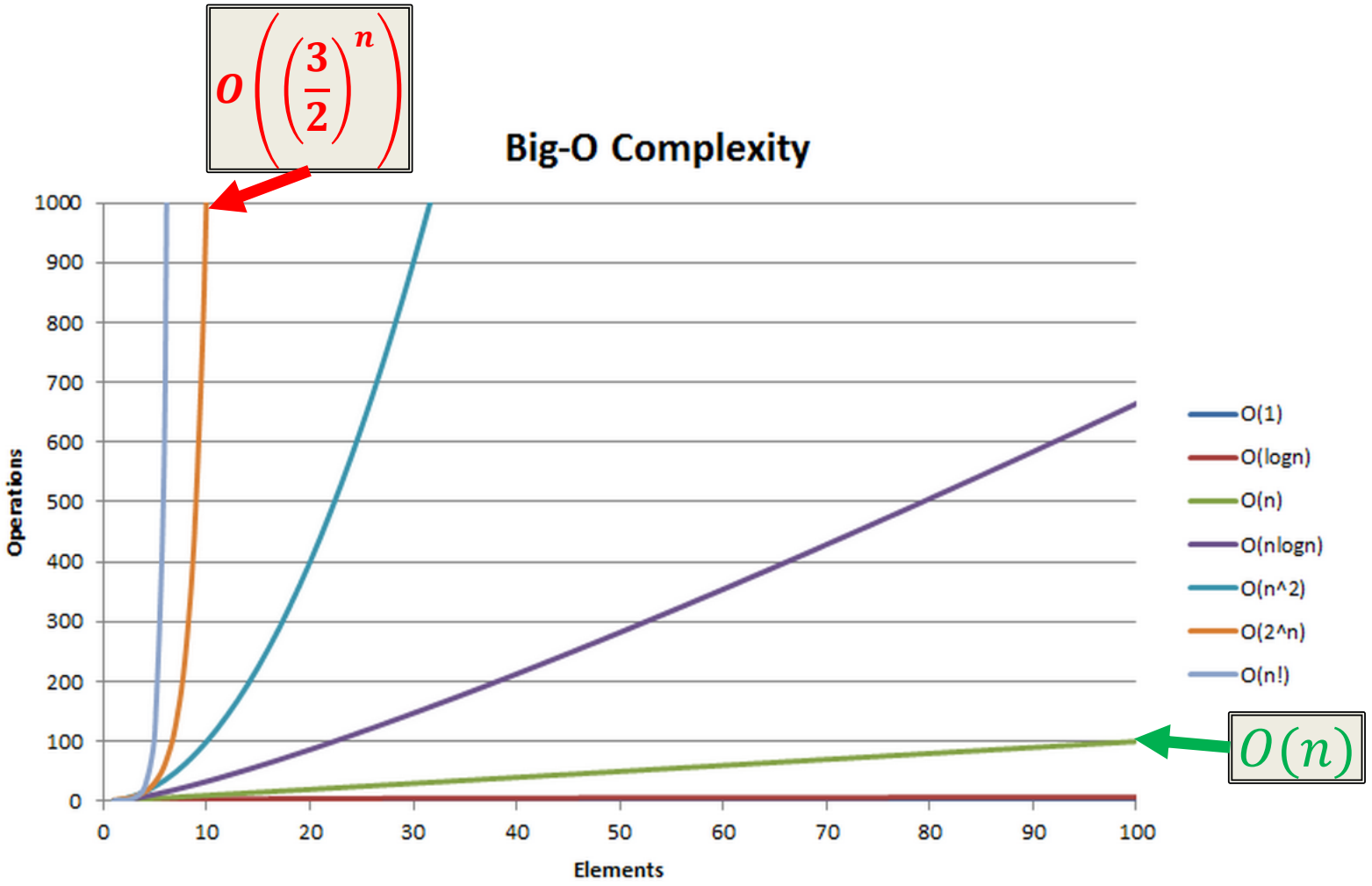
```
def f(n):
    if n==0: return 0
    if n==1: return 1
    if n>= 2: return f(n-1) + f(n-2)
```

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

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

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

Compute Fibonacci: Recursive vs. Non-recursive

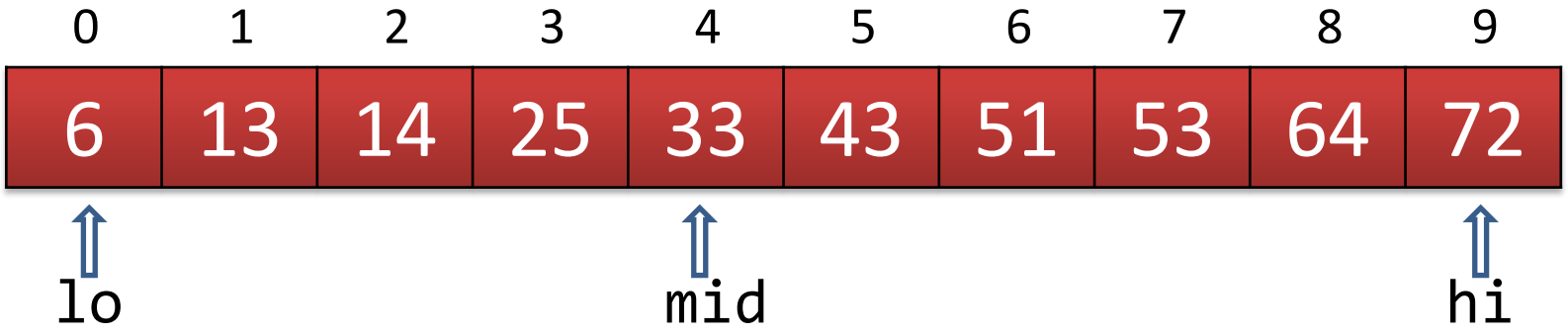


Example: Binary Search

- `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. Equal, return the index.
  4. Size <= 0, return -1.

Example: Binary Search




```
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 halve.
2. Bigger, search in the right halve.
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.

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

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

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
```

$5^6 = 5^3 \times 5^3$

$5^7 = 5^3 \times 5^3 \times 5$

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:

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

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

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

…

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 $k = \lg n$.

lg n = lg 2^k
lg n = k lg 2
lg n = k

We have:

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

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
```

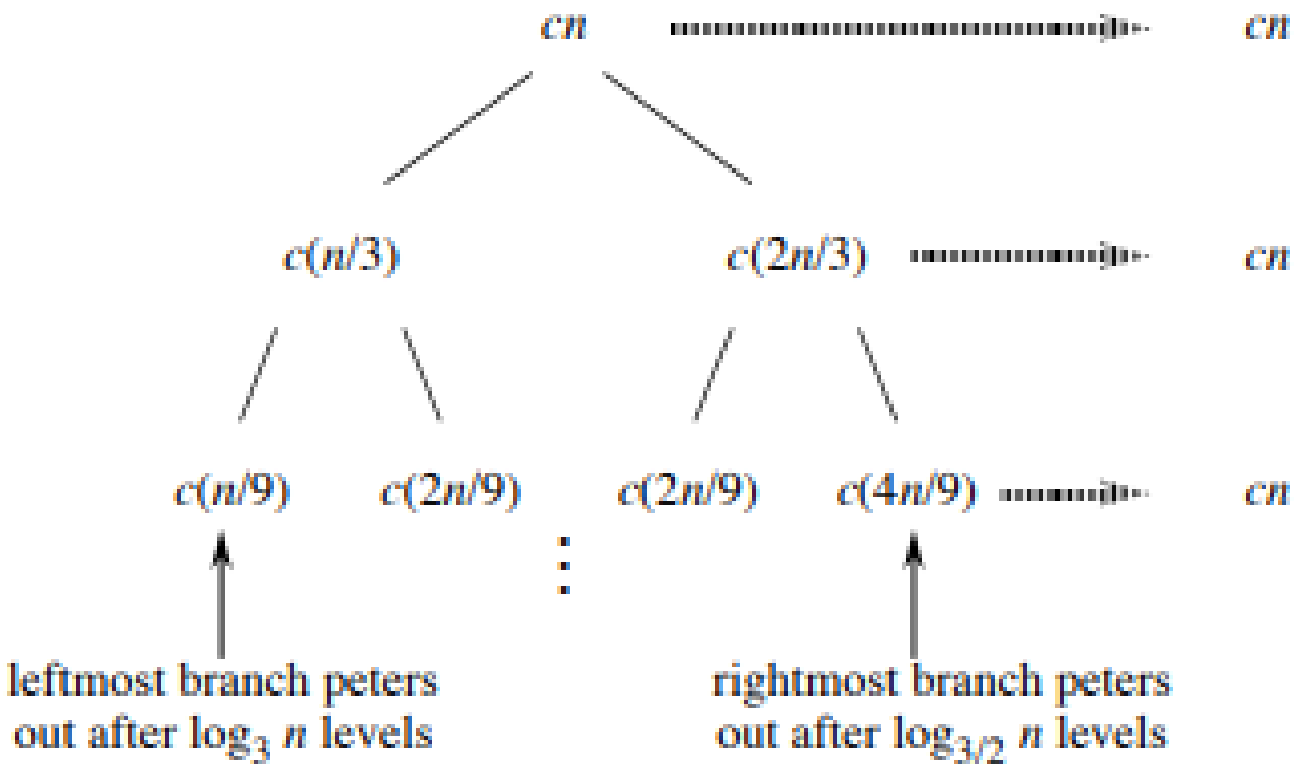
Time efficiency of the recursive power algorithm is of $O(\lg n)$.

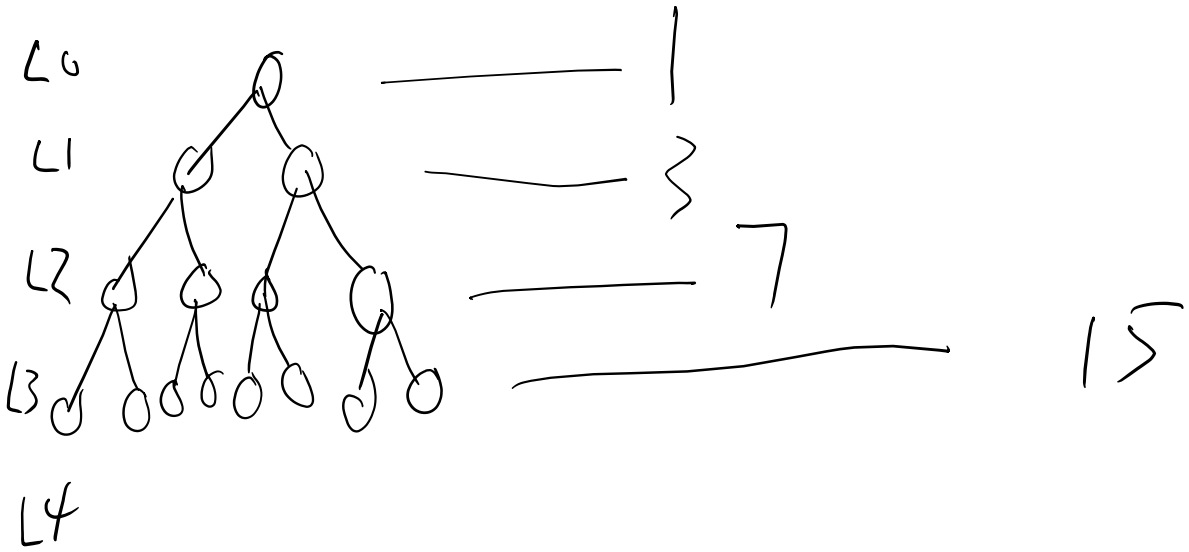
Recursion Tree I

- Each node represents a single subproblem.
- Sum the costs within each level.

Recursion Tree II

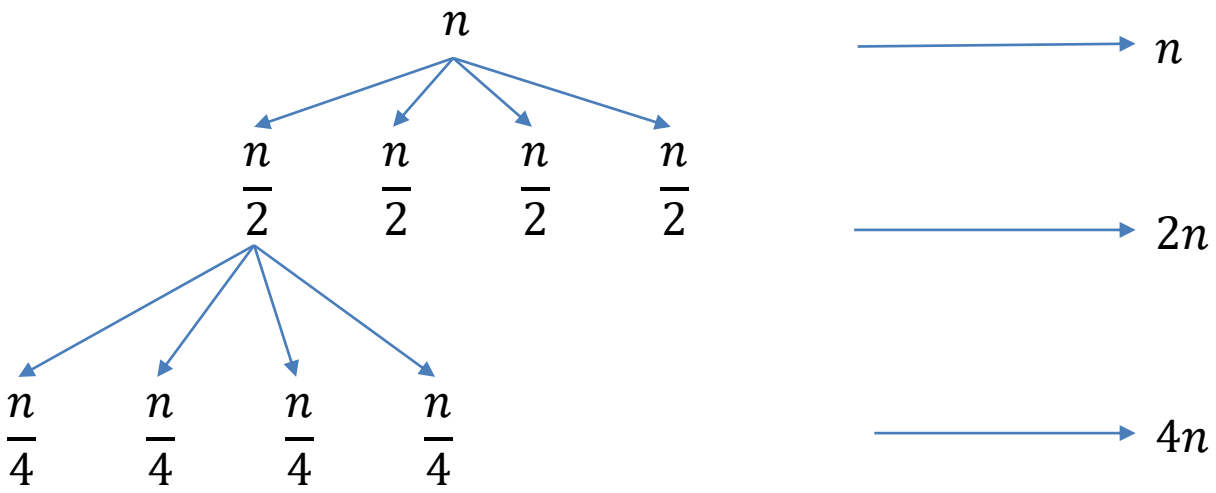
$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$




L0
L1
L2
L3
L4
1
3
7
15

Recursion Tree 1

- $T(n) = 4 * T\left(\frac{n}{2}\right) + n$



n + 2n + 4n + 8n =
n + 2^1n + 2^2n + 2^3n + …

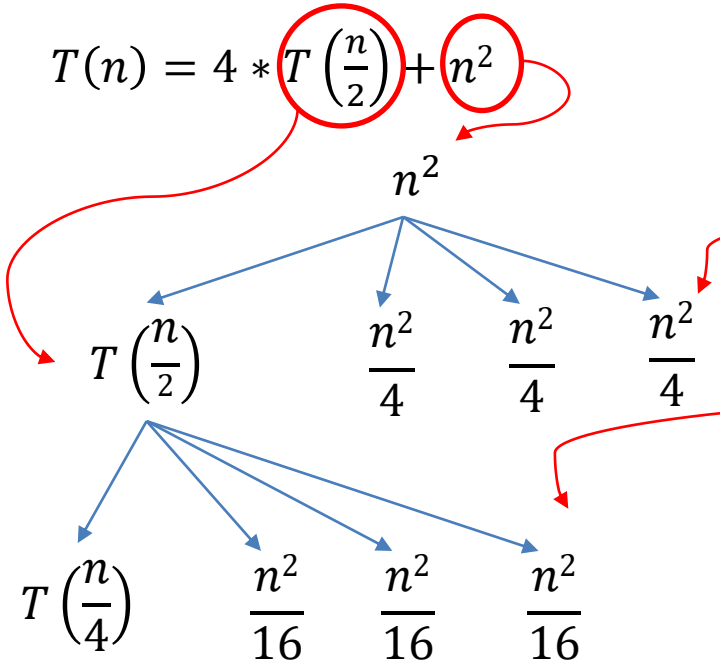
$$\sum_{k=0}^{n} ar^k = a + ar + \ldots + ar^n = \frac{a(r^{n+1} - 1)}{r - 1}$$

$= \frac{n(2^{\lg(n)} - 1)}{(2-1)}$ $= \frac{(n * 2^{\lg(n)} + n)}{1}$ = n * n

- a = first term (here is n)
- n = number of terms (here is lg n for tree)
- r = constant that each term is multiplied by (here is 2)

Recursion Tree 2

- $T(n) = 4 * T\left(\frac{n}{2}\right) + n^2$



$$T\left(\frac{n}{2}\right) = 4T\left(\frac{n/2}{2}\right) + \left(\frac{n}{2}\right)^2$$

$$T\left(\frac{n}{4}\right) = 4 * T\left(\frac{n/4}{2}\right) + \left(\frac{n}{4}\right)^2$$

$$T\left(\frac{n}{8}\right) = 4 * T\left(\frac{n/8}{2}\right) + \left(\frac{n}{8}\right)^2$$

$\rightarrow n^2$

$\rightarrow n^2$

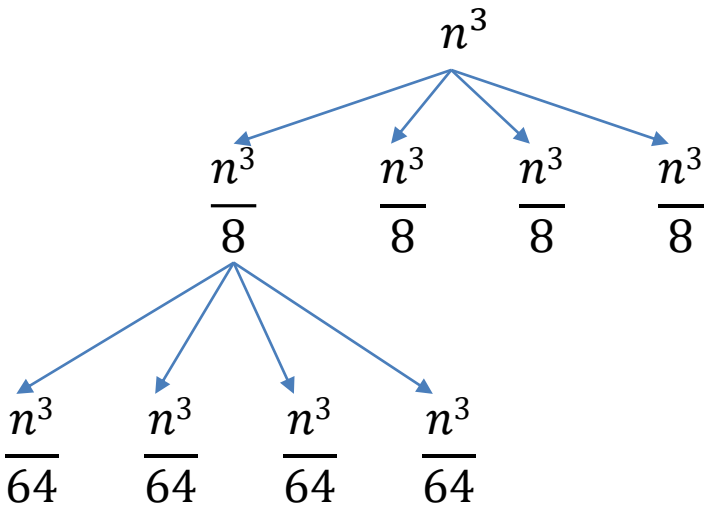
$\rightarrow n^2$

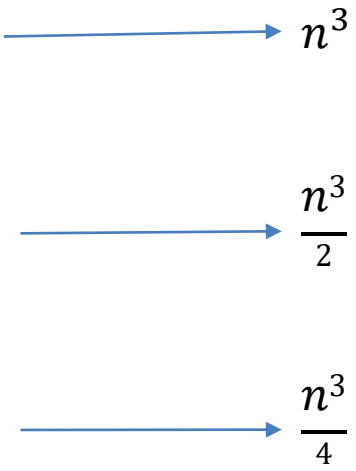
= n^2 * lg(n)

There are lg(n) levels of n^2

Recursion Tree 3

- $T(n) = 4 * T\left(\frac{n}{2}\right) + n^3$





$$= \frac{n^3(0.5^{\lg(n)} - 1)}{(0.5 - 1)}$$

$$= \frac{(n^3 - n^3 * 0.5^{\lg(n)})}{0.5}$$

$$= (2n^3 - 2 * n^3 * 0.5^{\lg(n)}) \quad = (2n^3 - 2n^2)$$

$n^3 + n^3/2^1 + n^3/2^2 + n^3/2^3 + \ldots$

$2^{\lg(n)} = n$

$2n^3/n = 2n^2$

Recurrences

- The running time of algorithms that recursively call themselves can often be described by a recurrence, i.e., an equation or inequality that describes a function in terms of its values on smaller inputs.
- Recursive algorithms typically follows a divide-and-conquer approach, which involves three steps at each level of recursion:
  1. Divide the problem into a number of sub-problems,
  2. Conquer the sub-problems by solving them recursively.
  3. Combine the solutions to sub-problems into the solution for the original problem.

Recurrences

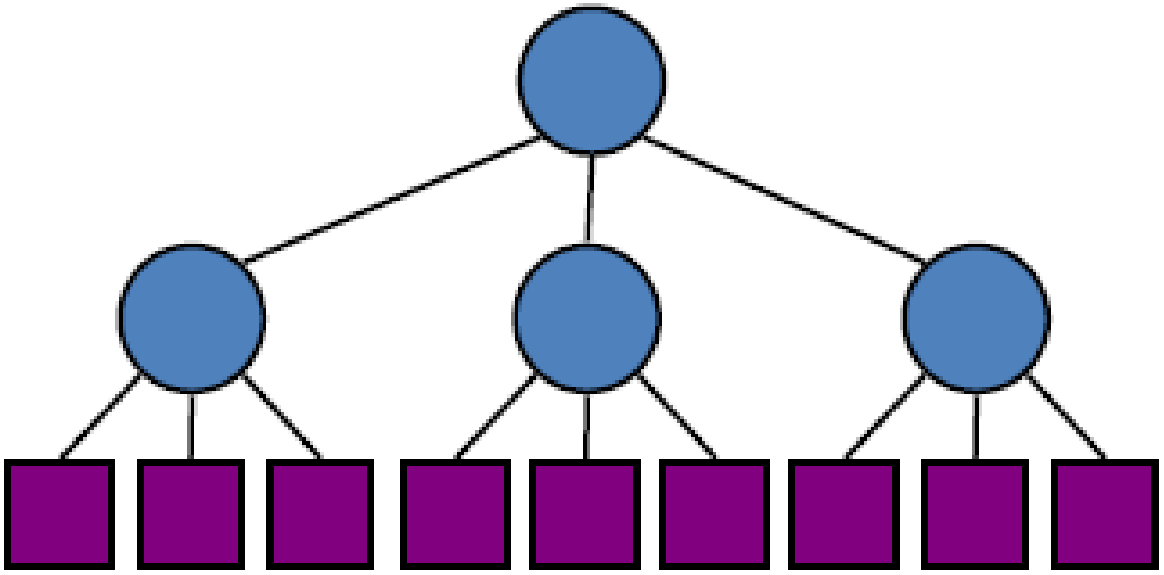
Analyzing Recursive Algorithms

- In a divide-and-conquer algorithm, if the problem size is small, the straightforward solution takes constant time, $\theta(1)$.
- Otherwise, division of the problem yields a sub-problems, each of which is 1/b the size of the original problem. Thus, the running time is given by:
- $$T(n) = \begin{cases} \theta(1) & if\ n \leq c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & otherwise \end{cases}$$
- where D(n) is the time to divide the problem into sub-problems and C(n) is the time to combine the solutions.

Recurrences

Analyzing Recursive Algorithms

- A schematic example of a divide-and-conquer algorithm in which at each step the problem is divided into three sub-problems is shown below:



Methods for solving Recurrences

- For solving recurrences (i.e., obtaining asymptotic θ or O bounds), we apply one of the following methods:

  1. Substitution Method: guesses a bound and then uses mathematical induction to show the guess is correct.
  2. Recursion-tree Method: converts the recurrence into a tree whose nodes represents the cost incurred at various levels of recursion.
  3. Master method: provides bounds for recurrences of the form $T(n) = aT(n/b) + f(n)$ for $a \geq 1, b > 1$

     where f(n) is a given function.

Substitution

- In substitution method, we observe two cases:
  - Case 1: For a given recurrence, we are required to prove that a given answer/solution is correct.
  - Case 2: For a given recurrence, we are required to find the answer/solution.
- In both cases, we can apply the mathematical induction (i.e., assuming that the given/guessed answer is correct for values smaller than n, then prove it is correct for n).
- Substitution method is powerful, but can be applied only in cases that we are given the answer, or it is easy to guess the form of the answer (upper or lower bounds).

Substitution

Example - Case 1:

- Using substitution, show that $T(n) = O(n \lg n)$ is the solution to recurrence $T(n) = 2\,T(n/2) + n$
- Proof: Let the formula be true for all values $k < n$. Thus,

$T(n) = 2\,(\,T(n/2)\,) + n$

$\leq 2\,[\,c \,.\, n/2 \,.\, \lg(n/2)\,] + n$

$= cn \lg(n/2) + n$

$= cn\,(\lg n) - n\,(c - 1)$

$\leq cn\,(\lg n)$ for $c \geq 1$

- Since O(nlgn) is the upper bound, $T(n) \leq cn\lg n$
- Sub. T(n/2) with c n lg n => $T(n/2)=c(n/2)\lg(n/2)$
- $\lg(n/2) = \lg n - \lg 2$
- $\lg 2 = 1$
- $cn\,(\lg n) - cn + n \leq cn\,(\lg n)$

- This completes the proof, in which, $T(n) = O(n \lg n)$.

Substitution

Example - Case 2:

- Using substitution method, find the solution to recurrence $T(n) = 2\,T(n/2) + n \lg n$

Guess #1: Let us guess that the solution is $T(n) = O(n \lg n)$ thus,

$T(n) = 2\,[\,T(n/2)\,] + n \lg n$

$\leq 2\,[\,c \cdot n/2 \cdot \lg(n/2)\,] + n \lg n$

$= cn \lg(n/2) + n \lg n$

$\leq cn \lg n - cn + n \lg n$

$= (c+1)n \lg n - cn$

n/2 n/2 n/2

$cn (\lg n) - cn + n \lg n$
$= (cn+n) \lg n - cn$
$= \mathbf{(c+1)}\, n \lg n - cn$
can never be $\leq \mathbf{c}\, n (\lg n)$

The guess is wrong:

- Since we cannot make this last equation to be less than $cn \lg n$.
- $[\,(c+1)n \lg n - cn\,]$ not $\leq$ $cn \lg n$

Substitution

Example - Case 2: (cont.)

- Using substitution method, find the solution to recurrence $T(n) = 2\,T(\lfloor n/2 \rfloor) + n \lg n$.

Guess #2: Let us guess that the solution is $T(n) = O(n^2)$. Thus,

$$
\begin{aligned}
T(n) &= 2\,[\,T(n/2)\,] + n \lg n \\
&\le 2\,[\,(c \,.\, (n/2)^2)\,] + n \lg n \\
&= \tfrac{1}{2}(cn^2) + n \lg n \\
&= cn^2 - \tfrac{1}{2}(cn^2) + n \lg n \\
&\le cn^2
\end{aligned}
$$

- The last equation is true for $c \ge 2$, and thus, $T(n) = O(n^2)$.

Substitution

SINGAPORE INSTITUTE OF TECHNOLOGY

Remarks on Guessing in Substitution Method

Remark: There is no general way to guess the correct solutions to recurrences.

- However, if a recurrence is similar to one you have seen before, then guessing a similar solution is reasonable. For example, $T(n) = 2\,T(n/2 + 17) + n$

  which is a similar to $T(n) = 2\,T(n/2) + n$

  asymptotically is bounded on $O(n \lg n)$.

- Another way to make a good guess is to loose upper and lower bounds on the recurrence and then reduce the range of uncertainty.

The Recursion-tree Method

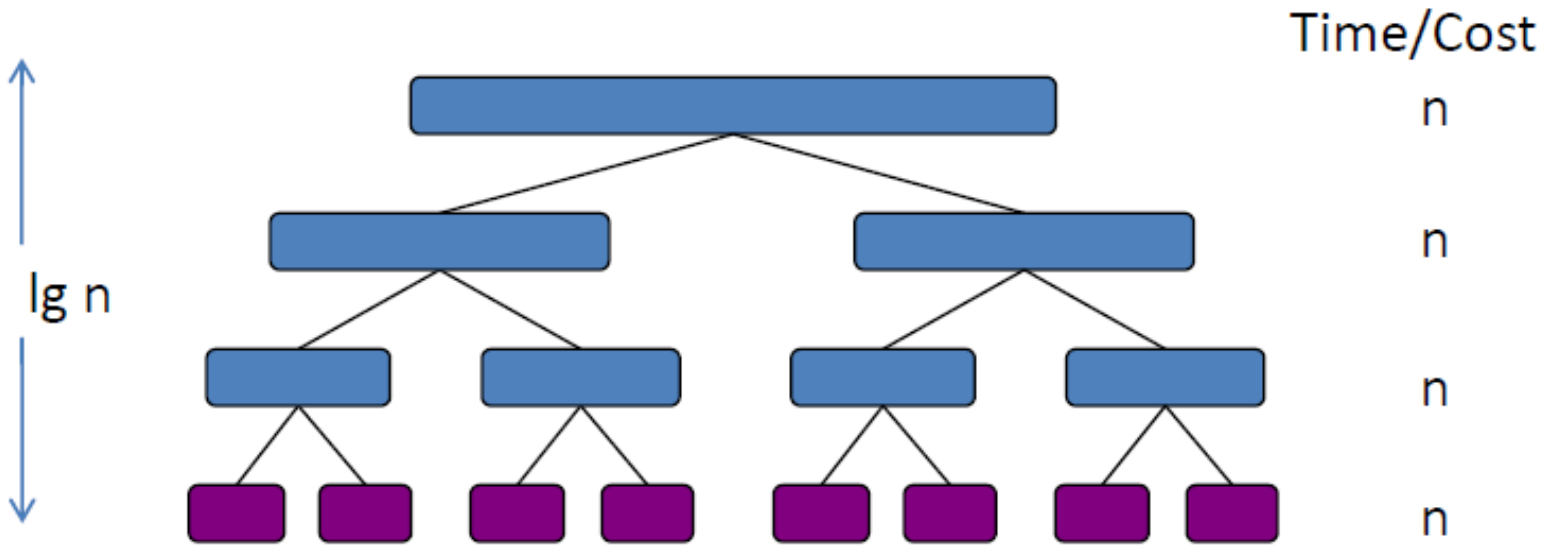
- A difficulty in using the substitution method is how to come up with a good guess.
- The recurrence tree solves this problem by showing the cost of each sub-problem.
- Recursion trees are particularly useful when the recurrence describes the running time of a divide-and-conquer algorithm.

The Recursion-tree Method

- Example:

  Find a solution to recurrence $T(n) = 2\,T(b/2) + n$.

- Create a recursion tree for the recurrence $T(n) = 2\,T(n/2) + n$



The running time of this recurrence is $(n \lg n)$.

The Master Method

- The master method (if it can be applied) can determine easily a solution to recurrences of the form

  $$T(n) = a \cdot T(n/b) + f(n)$$

  where $a \geq 1$ and $b > 1$ are constant and f(n) is an asymptotically positive function.

- In this recurrence, a sub-problems are solved recursively,
  - The problem is divided into 'a' sub-problems, each handles n/b number of data and requires time T(n/b), and
  - f(n) = D(n) + C(n) is the cost of dividing the problem and combining the results of the sub-problems.

B3. The Master Method

(The Master theorem)

$n^{(\log_b a)-\varepsilon} = \frac{n^{(\log_b a)}}{n^{\varepsilon}}$

$n^{(\log_b a)+\varepsilon} = n^{\varepsilon}\, n^{(\log_b a)}$

- Let a ≥ 1 and b > 1 be constants. The recurrence T(n) = a T(n/b) + f(n) can be bounded asymptotically as follows (3 cases):

Case	If	Then
1	f(n) = $O(n^{(\log_b a)-\varepsilon})$, for some constant ε > 0	T(n) = $\Theta(n^{\log_b a})$
2	f(n) = $\Theta(n^{\log_b a})$	T(n) = $\Theta(n^{\log_b a} \lg n)$
3	f(n) = $\Omega(n^{(\log_b a)+\varepsilon})$, for some constant ε > 0, and if a . f(n/b) ≤ c . f(n) for some constant c < 1 and all sufficiently large n	T(n) = Θ(f(n))

Det. f(n) or $n^{\log_b a}$ is the larger one
if $n^{\log_b a}$ is larger case 1, if f(n) is larger case 3, if both equal then case 2

B3. The Master Method

Remarks:

- To apply the master method some conditions must be satisfied.
- (Case 1) not only must f(n) < $n^{log_b a}$, it must be **polynomially smaller.**
- That is, f(n) must be asymptotically smaller than $n^{log_b a}$ by a factor of n^{ε} for some constant $\varepsilon > 0$.
- f(n) < $\frac{n^{(log_b a)}}{n^{\varepsilon}}$ if $\varepsilon=0$, then $n^0=1$, case 2; if $\varepsilon<0$, then case 3 $n^{(log_b a)+\varepsilon}$

1	f(n) = $O(n^{(log_b a)-\varepsilon})$, for some constant $\varepsilon > 0$	T(n) = $\Theta(n^{log_b a})$

B3. The Master Method

Remarks:

- To apply the master method some conditions must be satisfied.
- (Case 3) not only must f(n) > $n^{log_b a}$, it must be **polynomially larger** and
- in addition satisfy the regularity condition that a.f(n/b) ≤ c.f(n)

3	f(n) = $\Omega(n^{(log_b a)+\varepsilon})$, for some constant $\varepsilon > 0$, and a . f(n/b) ≤ c . f(n) for some constant c < 1 and all sufficiently large n	T(n) = Θ(f(n))

B3. The Master Method

Remarks:

- To apply the master method some conditions must be satisfied.
- The 3 cases don't cover all the possibilities for f(n).
- There is a gap between cases 1 and 2 when f(n) < $n^{log_b a}$ but not polynomially smaller.
- There is a gap between cases 2 and 3 when f(n) > $n^{log_b a}$ but not polynomially larger.
- If the function f(n) falls into one of these gaps, or if the regularity condition in case 3 fails to hold, the master method cannot be used to solve the recurrence.

- f(n)=1 and g(n)=n^2. Then f(n) is polynomially smaller than g(n)
- f(n)=$n^{1+\frac{1000}{\log n}}$ and g(n)=n^2. Then f(n) is polynomially smaller than g(n)
- f(n)=$\frac{n}{\log n}$ and g(n)=n. Then f(n) is not polynomially smaller nor polynomially larger than g(n)

$T(n) = 4T\left(\frac{n}{2}\right) + 1$ a=4, b=2, $n^{log_2 4}$=n^2 is polynomial larger than the constant function 1. This is case 1 of the master's theorem.

$T(n) = 4T\left(\frac{n}{2}\right) + \frac{n^2}{\log n}$ a=4, b=2, $\frac{n^2}{\log n}$ is not polynomially smaller nor polynomially larger than n^2. None of the three cases of master's theorem can be applied.

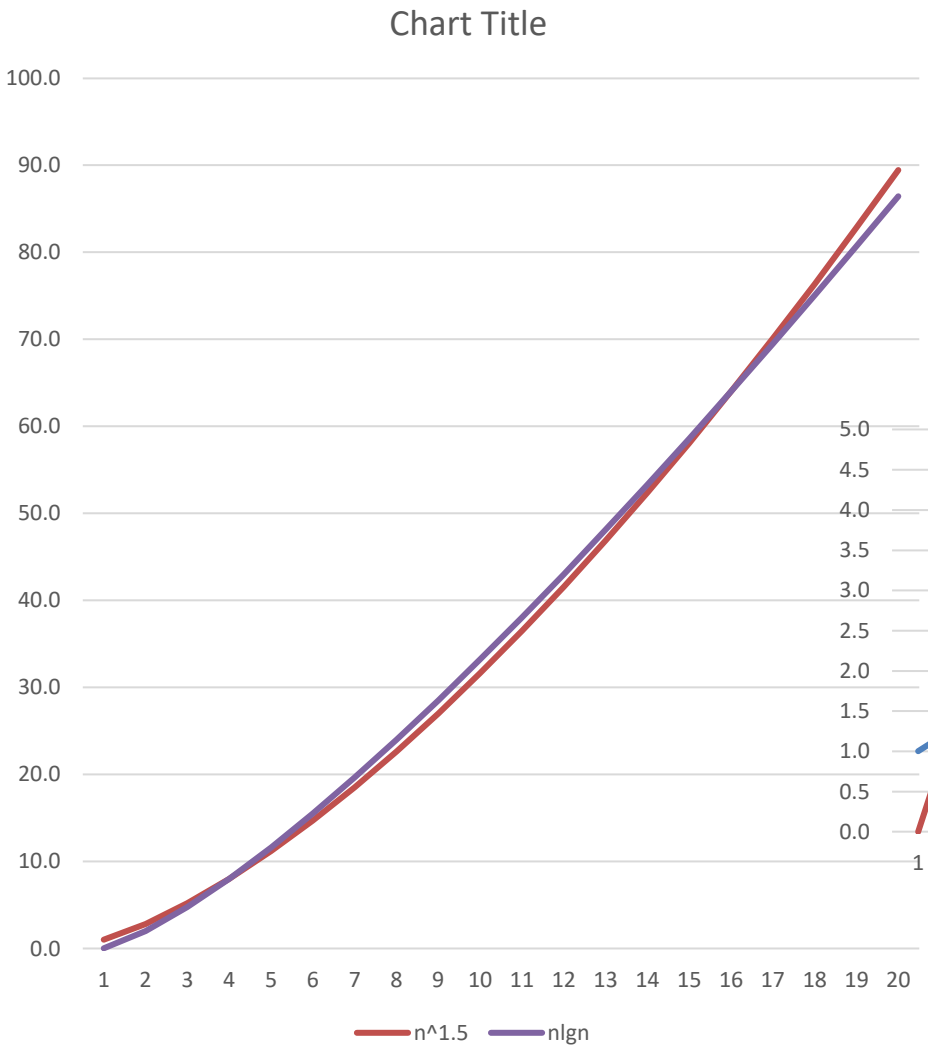


Chart Title
100.0
90.0
80.0
70.0
60.0
50.0
40.0
30.0
20.0
10.0
0.0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
n^1.5
nlgn

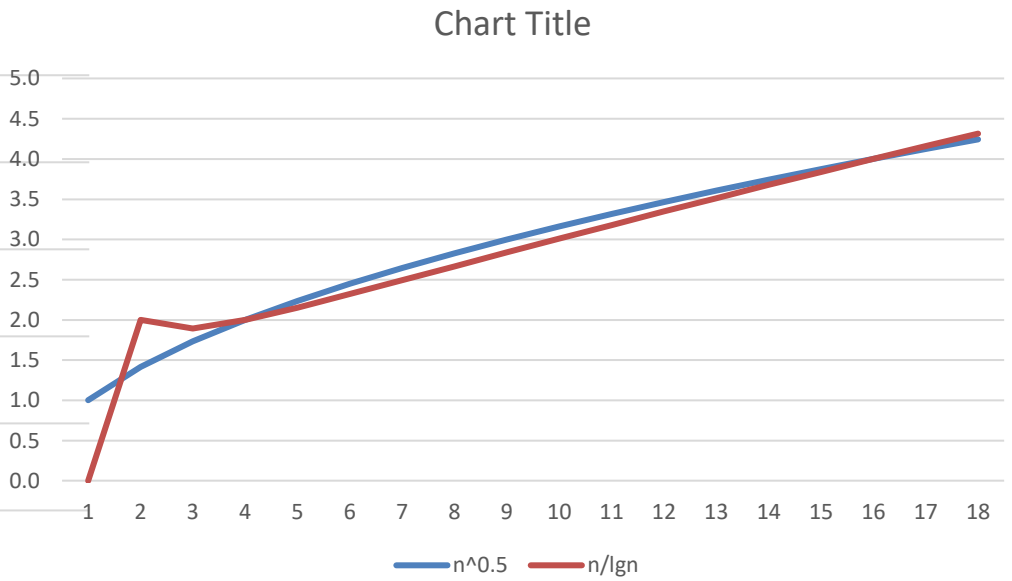


Chart Title
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
n^0.5
n/lgn

B3. The Master Method

Example 1:

$T(n) = a \cdot T(n/b) + f(n)$

- Solve the recurrence $T(n) = 9\,T(n/3) + n$
- For this recurrence, we have $a=9$, $b=3$, $f(n)=n$, and thus $n^{\log_b a} = n^{\log_3 9} = n^2$
- Since $n \leq n^{(2-\varepsilon)}$ for $0 < \varepsilon \leq 1$, (i.e. $n = n^{(2-1)}$) thus we apply case 1 which is $f(n) = O(n^{(\log_b a)-\varepsilon})$, where $0 < \varepsilon \leq 1$
- Thus, the solution to this recurrence is $T(n) = \Theta(n^2)$.

1	$f(n) = O(n^{(\log_b a)-\varepsilon})$, for some constant $\varepsilon > 0$	$T(n) = \Theta(n^{\log_b a})$
2	$f(n) = \Theta(n^{\log_b a})$	$T(n) = \Theta(n^{\log_b a} \lg n)$
3	$f(n) = \Omega(n^{(\log_b a)+\varepsilon})$, for some constant $\varepsilon > 0$, and if $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$ and all sufficiently large n	$T(n) = \Theta(f(n))$

B3. The Master Method

$T(n) = a \cdot T(n/b) + f(n)$

Example 2:

- Solve the recurrence $T(n) = T(2n/3) + 1$
- For this recurrence, we have $a=1$, $b=3/2$, $f(n)=1$, and thus $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$
- Since $f(n) = \Theta(n^{(\log_b a)}) = \Theta(1)$, case 2 applies
- Thus, the solution to this recurrence is $T(n) = \Theta(1 \times \lg n)$.

1	$f(n) = O(n^{(\log_b a)-\varepsilon})$, for some constant $\varepsilon > 0$	$T(n) = \Theta(n^{\log_b a})$
2	$f(n) = \Theta(n^{\log_b a})$	$T(n) = \Theta(n^{\log_b a} \lg n)$
3	$f(n) = \Omega(n^{(\log_b a)+\varepsilon})$, for some constant $\varepsilon > 0$, and if $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$ and all sufficiently large n	$T(n) = \Theta(f(n))$

B3. The Master Method

$T(n) = a \cdot T(n/b) + f(n)$

Example 3:

- Solve the recurrence $T(n) = 3\,T(n/4) + n \lg n$
- For this recurrence, we have $a=3$, $b=4$, $f(n)=n \lg n$, and thus $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$
- Since $n \lg n > n^{0.793}$, $f(n) = \Omega(n^{(\log_4 3)+\varepsilon})$, $\varepsilon \approx 0.2$ (i.e. <0.207)
- Case 3 applies if we can show that the regularity condition holds for f(n). For sufficiently large n

$$a.f(n/b) = 3.(n/4) \lg(n/4) \leq (3/4)\, n \lg n = c.f(n) \quad \text{for } c=3/4$$

Therefore the solution is $T(n) = \Theta(f(n)) = \Theta(n \lg n)$

3	$f(n) = \Omega(n^{(\log_b a)+\varepsilon})$, for some constant $\varepsilon > 0$, and if $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$ and all sufficiently large n	$T(n) = \Theta(f(n))$

B3. The Master Method

$T(n) = a \cdot T(n/b) + f(n)$

Example 4:

- Solve the recurrence $T(n) = 2\,T(n/2) + n \lg n$
- For this recurrence, we have $a=2$, $b=2$, $f(n)=n \lg n$, and thus $n^{\log_b a} = n^{\log_2 2} = n^1 = n$
- Although $f(n) = n \lg n$ is asymptotically larger than n, it is not polynomially larger (because $\lg n$ is asymptotically smaller than n^{ε} for any positive constant ε), therefore we cannot use case 3 of the Master method.
- Hence, the recurrence falls into the gap between case 2 and case 3 (i.e. The master method cannot be applied).

3	$f(n) = \Omega(n^{(\log_b a)+\varepsilon})$, for some constant $\varepsilon > 0$, and if $a \cdot f(n/b) \le c \cdot f(n)$ for some constant $c < 1$ and all sufficiently large n	$T(n) = \Theta(f(n))$

- end