Complexity

Instructor: Meng-Fen Chiang

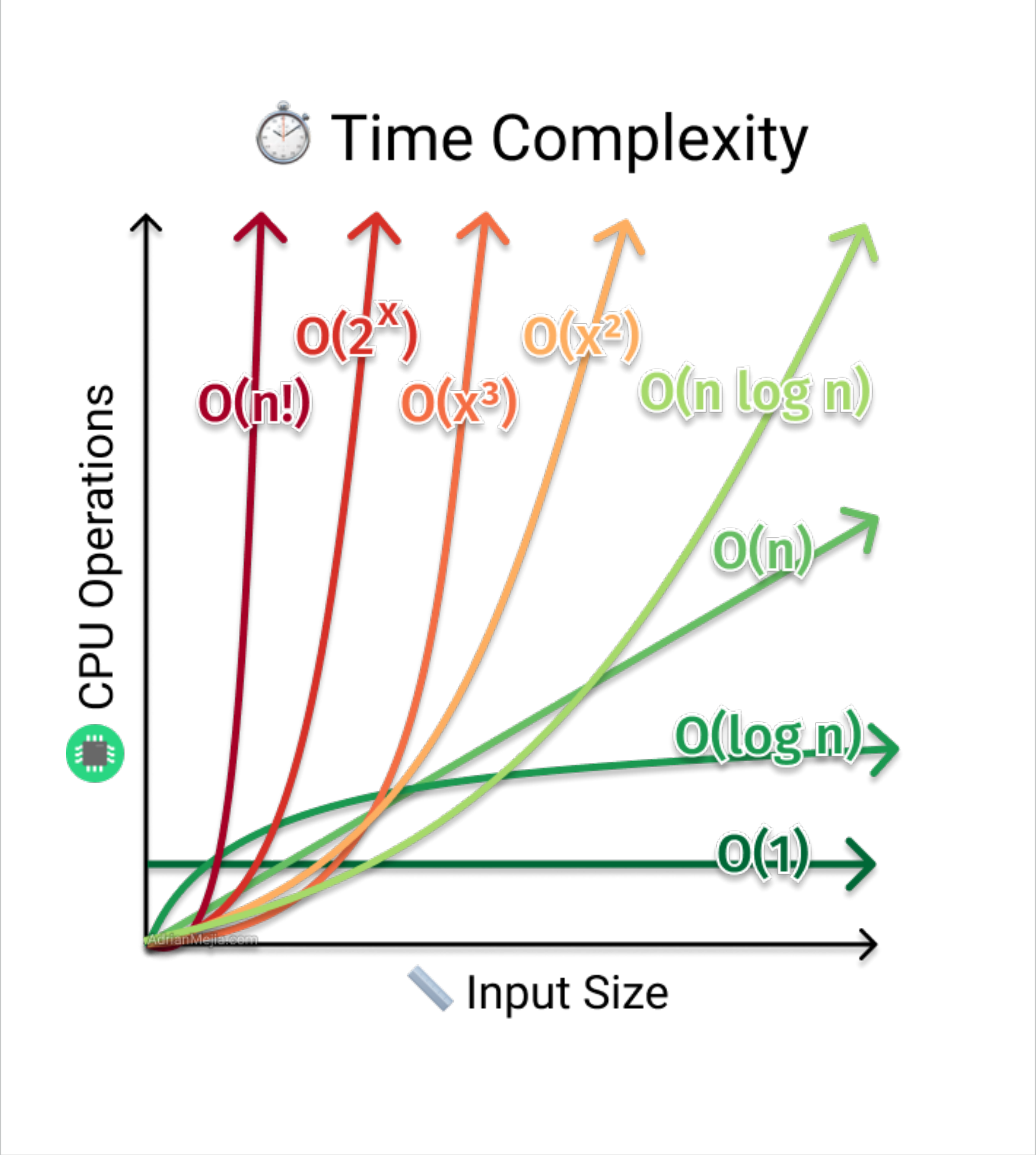
COMPCSI220: WEEK 8

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

Slides adapted from Mark Wilson, Georgy Gimel'farb, Simone Linz and Tanya Gvozdeva

OUTLINE

- Analysis of Algorithm
  - Illustrative examples
- Time Complexity
  - How to measure running time?
  - Illustrative examples
- Asymptotic Notation



Analysis of Algorithms

- What to analyze
  - Domain of definition – what inputs are legal?
  - Correctness – does it solve the problem for all legal inputs?

- Efficiency: its maximum or average requirements to resources:
  - Runtime
  - Memory space
  - Other resources

- There could be different implementations of the same algorithm: different programs, programming languages, computer platforms, operating systems, etc.
- The analysis should be isolated from a particular implementation.

Complexity

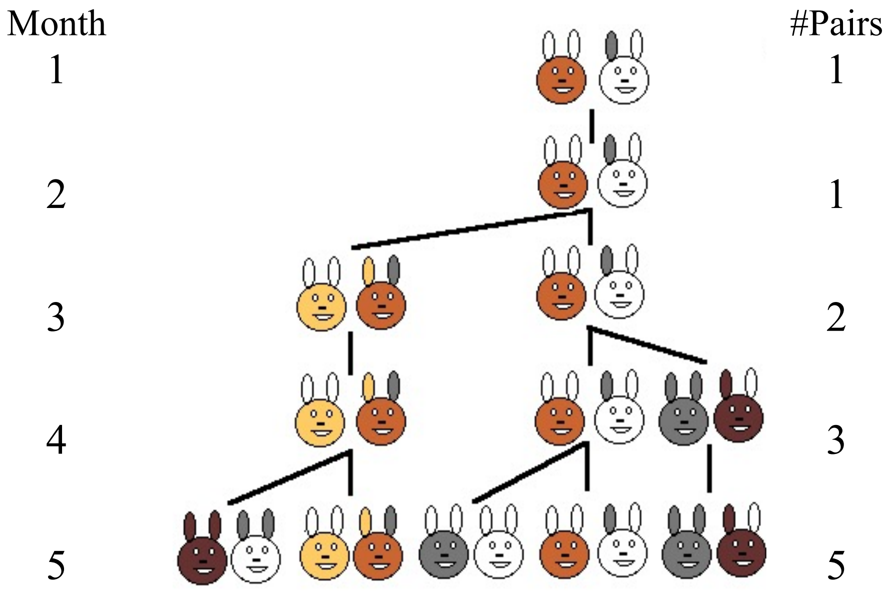
- In a practical sense, complexity is an estimate of resource requirement
  - How long would it take to drive from Auckland to Wellington?
  - How much petrol would we need to drive from Auckland to Wellington?

Complexity (Contd.)

- Time complexity
    - An estimate of how long it would take to complete a task
- Space complexity
    - An estimate of how much memory the task might require for completion

Example: Fibonacci Numbers

- Italian mathematician, Leonardo Fibonacci (1170–1250). A problem of breeding rabbits.
  - A pair of rabbits takes a month to become mature and start to have pairs of baby rabbits
  - Each pair of newly born rabbits also take a month to reach maturity.
  - How many pairs of rabbits, $F(n)$ would there be after n months?



Fibonacci numbers:

$$F(n) = F(n-1) + F(n-2)$$
$$F(0) = 0, \quad F(1) = 1$$

This immediately suggests a recursive algorithm

http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibnat.html

Algorithm 1 Slow method for computing Fibonacci numbers

1: **function** SLOWFIB(integer n)

2: **if** n < 0 **then return** 0

3: **else if** n = 0 **then return** 0

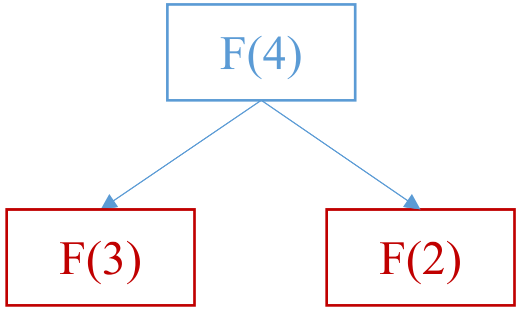
4: **else if** n = 1 **then return** 1

5: **else return** SLOWFIB($n - 1$) + SLOWFIB($n - 2$)

Correctness: The algorithm SLOWFIB is obviously correct

Efficiency: This algorithm is not efficient! It does a lot of repeated computation

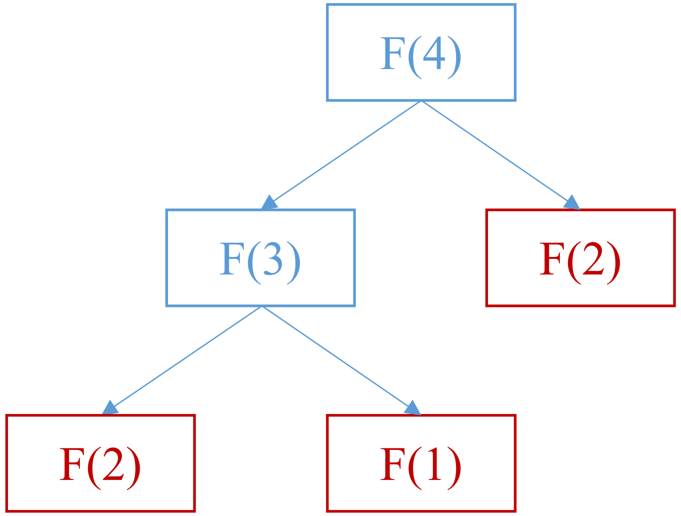
Example – F(4)



Algorithm 1 Slow method for computing Fibonacci numbers

```
function SLOWFIB(integer n)
    if n < 0 then return 0
    else if n = 0 then return 0
    else if n = 1 then return 1
    else return SLOWFIB(n − 1) + SLOWFIB(n − 2)
```

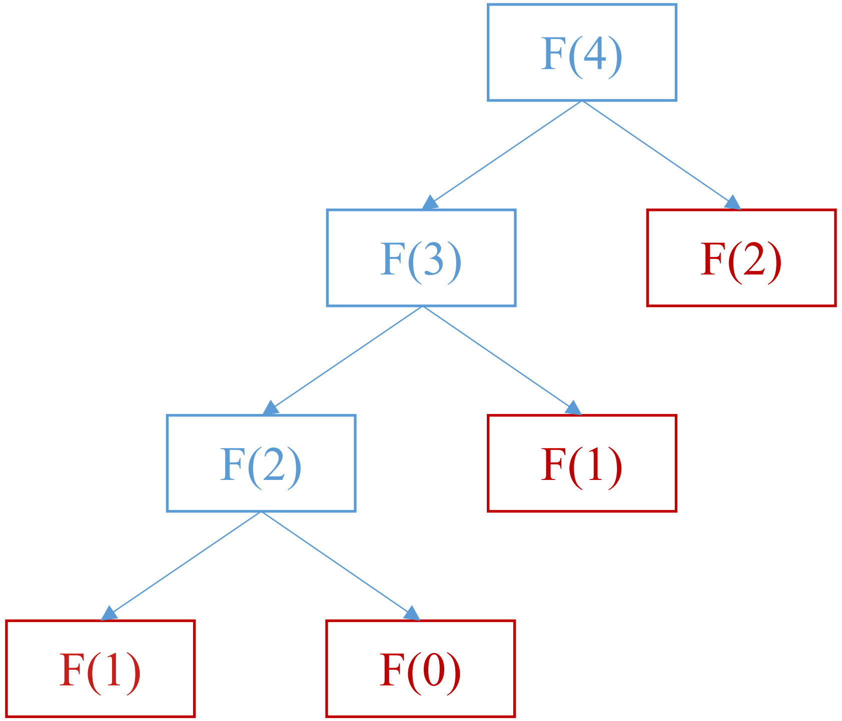
Example – F(4)



Algorithm 1 Slow method for computing Fibonacci numbers

```
function SLOWFIB(integer n)
    if n < 0 then return 0
    else if n = 0 then return 0
    else if n = 1 then return 1
    else return SLOWFIB(n − 1) + SLOWFIB(n − 2)
```

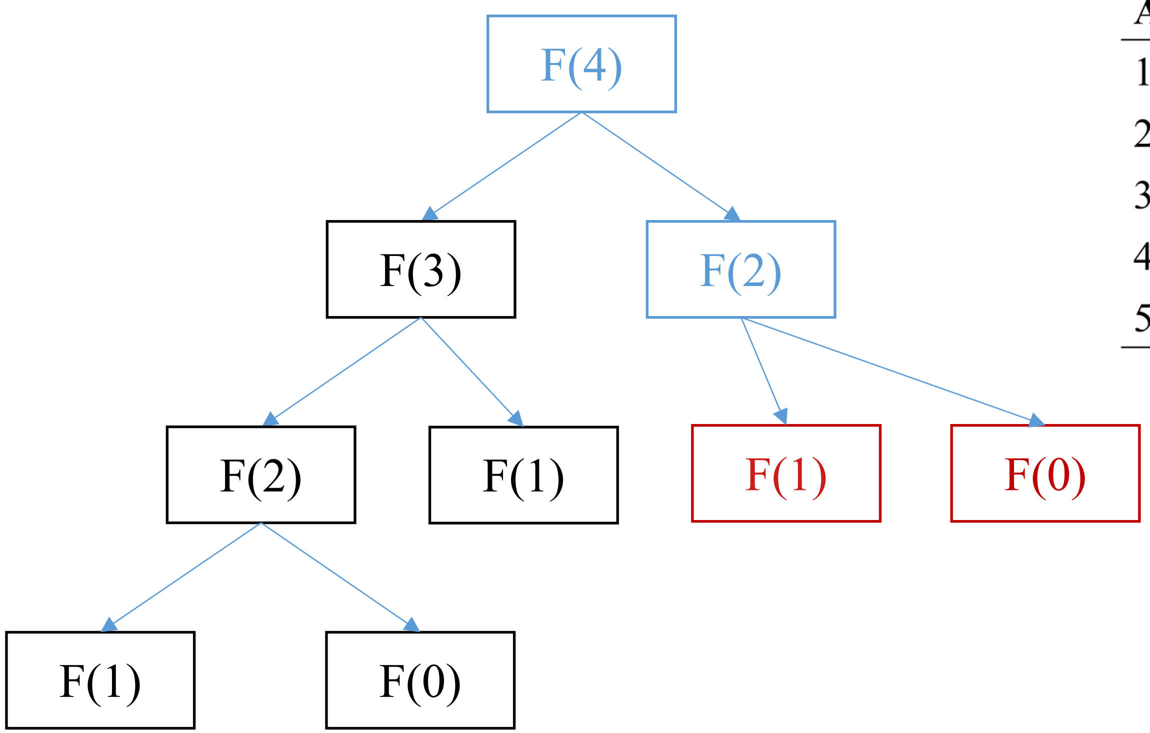
Example – F(4)



Algorithm 1 Slow method for computing Fibonacci numbers

1: **function** SLOWFIB(integer n)
2: **if** n < 0 **then return** 0
3: **else if** n = 0 **then return** 0
4: **else if** n = 1 **then return** 1
5: **else return** SLOWFIB($n - 1$) + SLOWFIB($n - 2$)

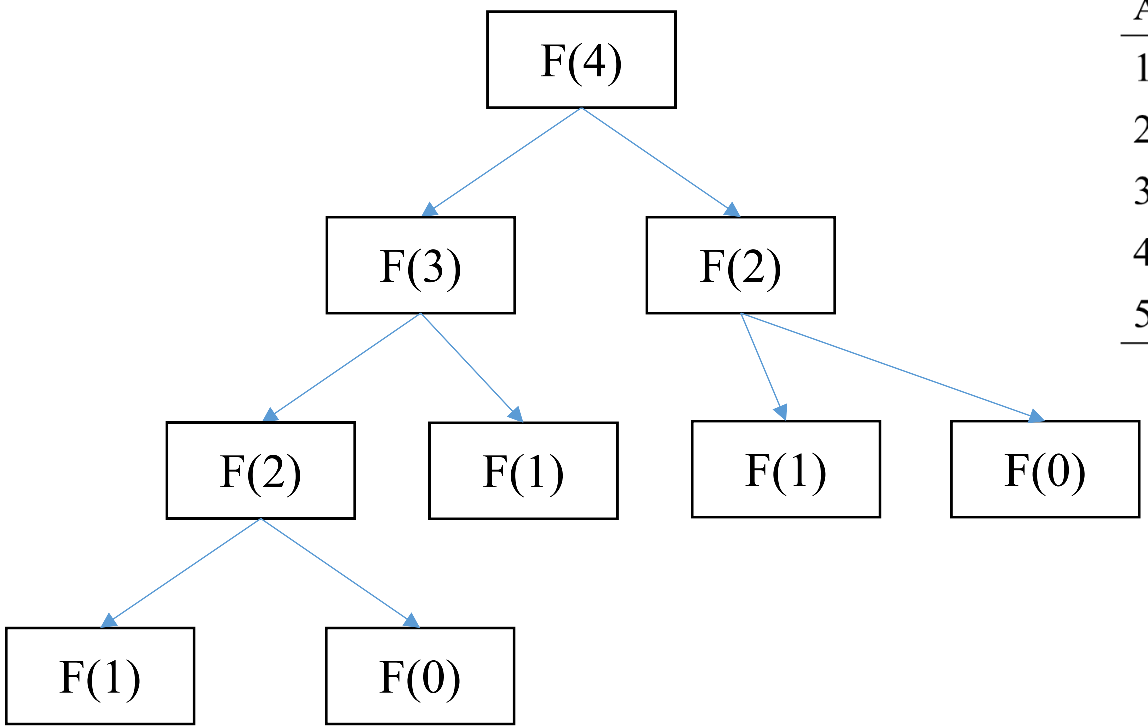
Example – F(4)



Algorithm 1 Slow method for computing Fibonacci numbers

```
function SLOWFIB(integer n)
    if n < 0 then return 0
    else if n = 0 then return 0
    else if n = 1 then return 1
    else return SLOWFIB(n − 1) + SLOWFIB(n − 2)
```

Example – F(4)



Algorithm 1 Slow method for computing Fibonacci numbers

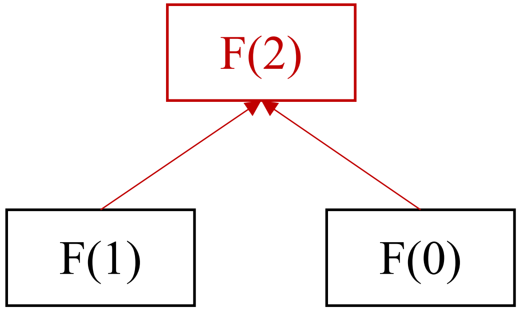
```
function SLOWFIB(integer n)
    if n < 0 then return 0
    else if n = 0 then return 0
    else if n = 1 then return 1
    else return SLOWFIB(n − 1) + SLOWFIB(n − 2)
```

- With a small (fixed) amount of extra space, we can do better, by working from the bottom up instead of from the top down.

Algorithm 2 Fast method for computing Fibonacci numbers

```
function FASTFIB(integer n)
    if n < 0 then return 0
    else if n = 0 then return 0
    else if n = 1 then return 1
    else
        a ← 1                    ▷ stores F(i − 1) at bottom of loop
        b ← 0                    ▷ stores F(i − 2) at bottom of loop
        for i ← 2 to n do
            t ← a
            a ← a + b
            b ← t
    return a
```

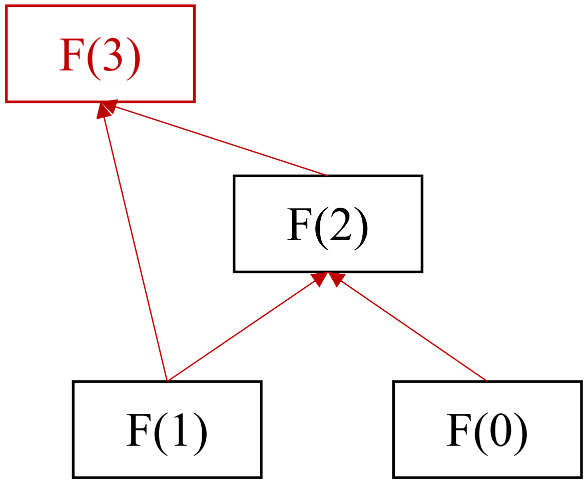
Example – F(4)



Algorithm 2 Fast method for computing Fibonacci numbers

```
function FASTFIB(integer n)
    if n < 0 then return 0
    else if n = 0 then return 0
    else if n = 1 then return 1
    else
        a ← 1                    ▷ stores F(i − 1) at bottom of loop
        b ← 0                    ▷ stores F(i − 2) at bottom of loop
        for i ← 2 to n do
            t ← a
            a ← a + b
            b ← t
    return a
```

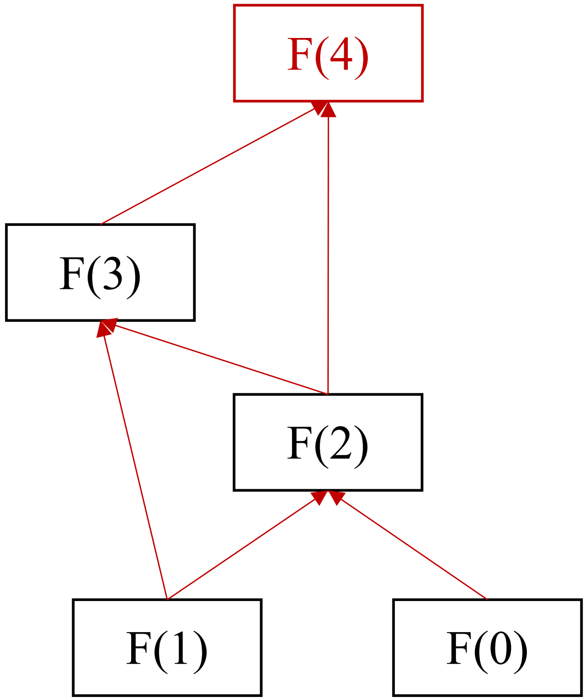
Example – F(4)



Algorithm 2 Fast method for computing Fibonacci numbers

```
function FASTFIB(integer n)
    if n < 0 then return 0
    else if n = 0 then return 0
    else if n = 1 then return 1
    else
        a ← 1                          ▷ stores F(i − 1) at bottom of loop
        b ← 0                          ▷ stores F(i − 2) at bottom of loop
        for i ← 2 to n do
            t ← a
            a ← a + b
            b ← t
    return a
```

Example – F(4)

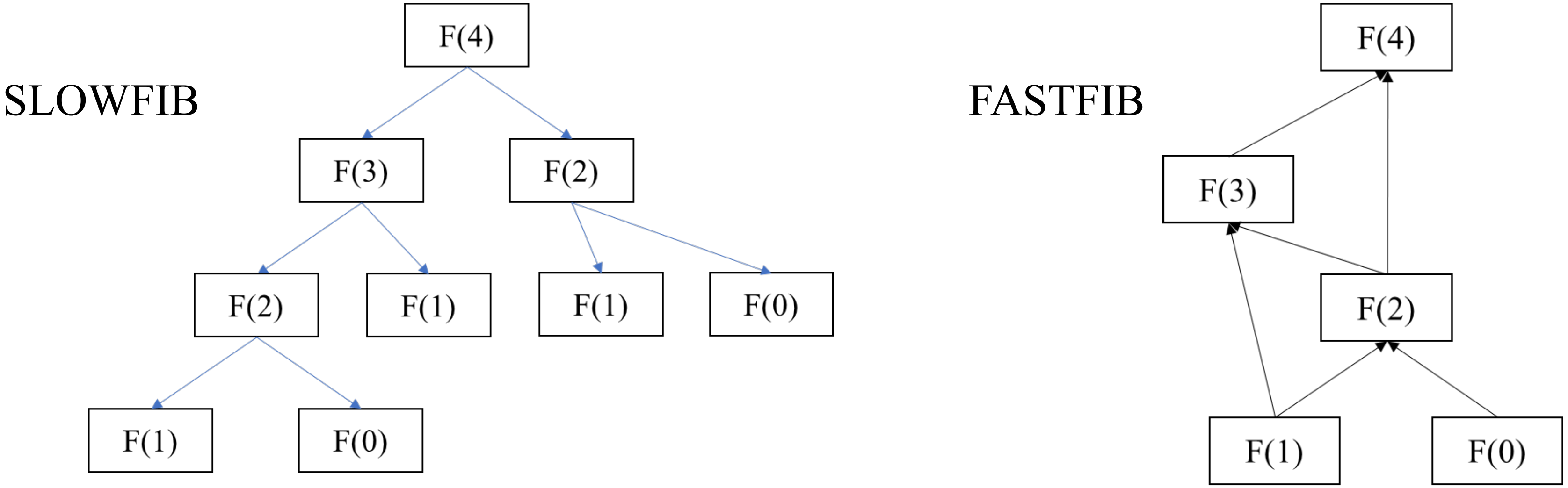


Algorithm 2 Fast method for computing Fibonacci numbers

function FASTFIB(integer n)
    if n < 0 **then return** 0
    else if n = 0 **then return** 0
    else if n = 1 **then return** 1
    else
        $a \leftarrow 1$ ▷ stores $F(i-1)$ at bottom of loop
        b $\leftarrow 0$ ▷ stores $F(i-2)$ at bottom of loop
        for $i \leftarrow 2$ **to** n **do**
            $t \leftarrow a$
            $a \leftarrow a + b$
            $b \leftarrow t$
    return a

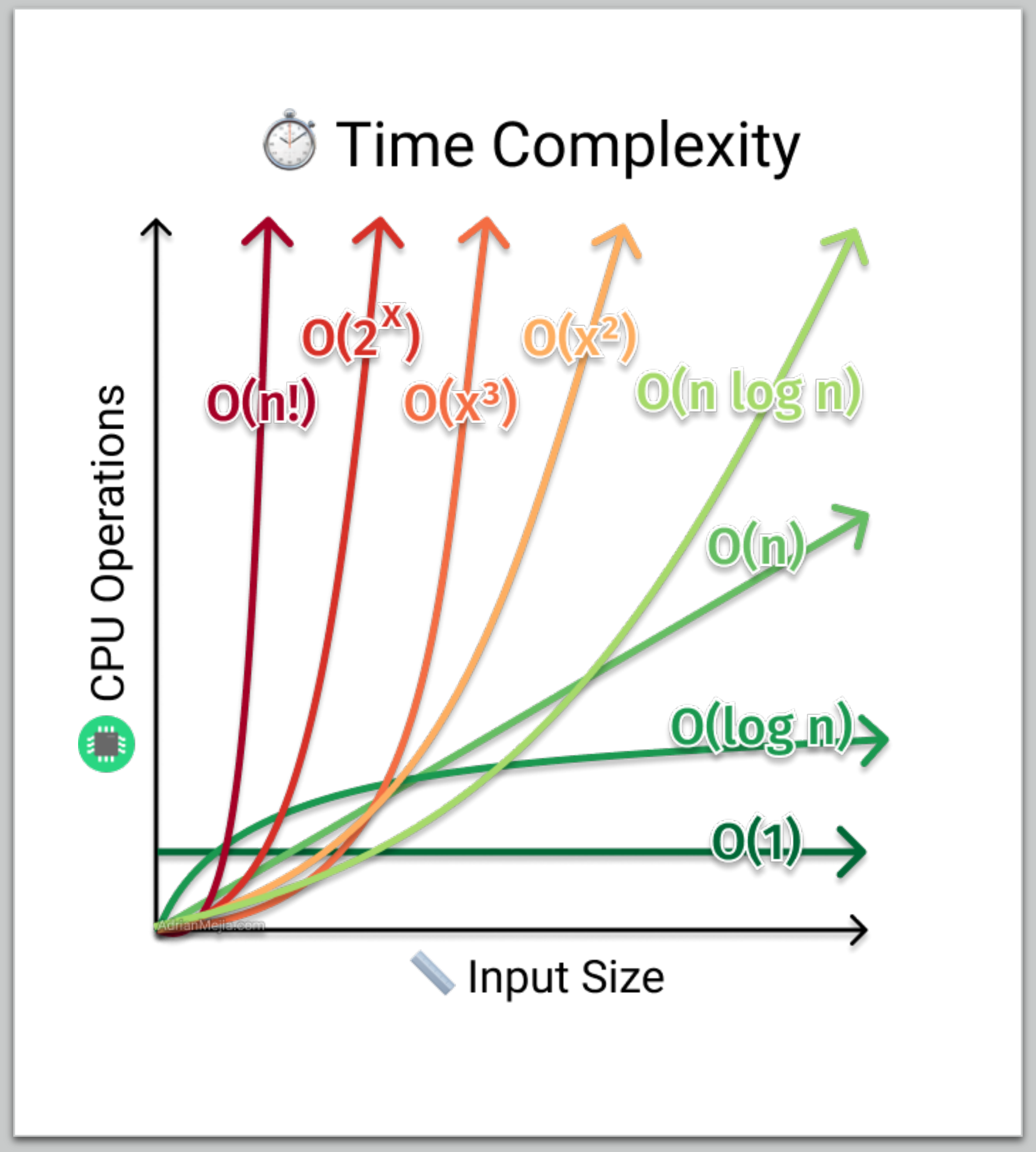
Analysis of the Fast Algorithm

- It is easy to see that the number of additions, function calls, etc needed by **FASTFIB** to compute $F(n)$ has the form $An + B$ for some constants A,B.



SUMMARY

- Analysis of Algorithm: Fibonacci numbers
    - Slow implementation
    - Fast implementation
- **Time Complexity**
    - **How to measure running time?**
    - **Illustrative examples**
- Asymptotic Notation



Time Complexity

- The actual running time of an algorithm $\mathcal{A}$ on a given input X depends on implementation, such as programming languages used, operating systems and hardware.

- The running time usually grows with the size of the input. Running time for very small inputs is not usually important; it is large inputs that cause problems if the algorithm is inefficient.

Elementary Operations

- We use the concept of elementary operation as our basic measuring unit of running time. This is any operation whose execution time does NOT depend on the size of the input.

- Typically, a standard unary or binary arithmetic operation:
  - Negation (-5)
  - Addition / subtraction ($5 + 37$; $350 - 75$)
  - Multiplication / division / modulo (67×89; $399/54$; $399\%54$)
  - Boolean operations (x AND y; x OR y, etc.)
  - Binary comparisons ($x == y$; $x \le y$; $x < y$; $x \ge y$; etc.)
  - variable assignments, function calls.

- The running time $T(n)$ of algorithm $\mathcal{A}$ on input X of size n is the number of elementary operations used when X is fed into $\mathcal{A}$.

Time Complexity

Running time $T(n)$		Input size of n			
Function	*Notation*	10	100	1000	10^7
Constant	1	1	1	1	1
Logarithmic	$\log n$	1	2	3	7
Linear	n	1	10	100	10^6
"Linearithmic"	$n \log n$	1	20	300	7×10^6
Quadratic	n^2	1	100	10000	10^{12}
Cubic	n^3	1	1000	10^6	10^{18}
Exponential	2^n	1	10^{27}	10^{298}	$10^{3010296}$

Constant Time Complexity

- Algorithm: Swapping two elements
- This is a constant time algorithm

Algorithm 1 Swapping two elements in an array

Require: $0 \leq i \leq j \leq n - 1$

    function SWAP(array $a[0 \ldots n-1]$, integer i, integer j)

        $t \leftarrow a[i]$

        $a[i] \leftarrow a[j]$

        $a[j] \leftarrow t$

        return a

Linear Time Complexity

- Algorithm: finding the maximum value
- This is a linear time algorithm, since it makes one pass through the array and does a constant amount of work each time.

Algorithm 2 Finding the maximum in an array

1: **function** FINDMAX(array $a[0 \ldots n-1]$)
2: $k \leftarrow 0$ ▷ location of maximum so far
3: **for** $j \leftarrow 1$ *to* $n-1$ **do**
4: **if** $a[k] < a[j]$ **then**
5: $k = j$
6: **return** k

Logarithmic Time Complexity

- Algorithm: : loop increments
- This runs in logarithmic time because i doubles about log n times until reaching n.

Algorithm 3 Example: exponential change of variable in loop

$i \leftarrow 1$
while $i \leq n$ **do**
    $i \leftarrow 2 * i$
    print i

Quadratic Time Complexity

- Algorithm: nested loops
- This runs in quadratic time, because the inner loop runs $n-i+1$ times, while the outer loop runs n times. That is:

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

Algorithm 4 Snippet: Nested loops

1: **for** $i \leftarrow 1$ to n **do**
2:     **for** $j \leftarrow i$ to n **do**
3:         **print** $i + j$

Exponential Time Complexity

- Algorithm: SLOWFIB
- SLOWFIB makes $F(n)$ function calls each of which involves a constant number of elementary operations. It turns out that $F(n)$ grows exponentially in n, so this is an exponential time algorithm.

Algorithm 5 Slow method for computing Fibonacci numbers

```
1: function SLOWFIB(integer n)
2:     if n < 0 then return 0
3:     else if n = 0 then return 0
4:     else if n = 1 then return 1
5:     else return SLOWFIB(n − 1) + SLOWFIB(n − 2)
```

Example: Time Complexity Analysis

- Algorithm: Sum of an array $s = \sum_{i=0}^{n-1} a[i]$
- Summing n elements of the array a repeats elementary fetch/add operations n times.

Algorithm 6 Summing elements of an array $a[0 \dots n-1]$

```
Require: array a
    function SUM(a[0 … n − 1])
        s ← 0
        for i ← 0 to n − 1 do
            s ← s + a[i]
        return s
```

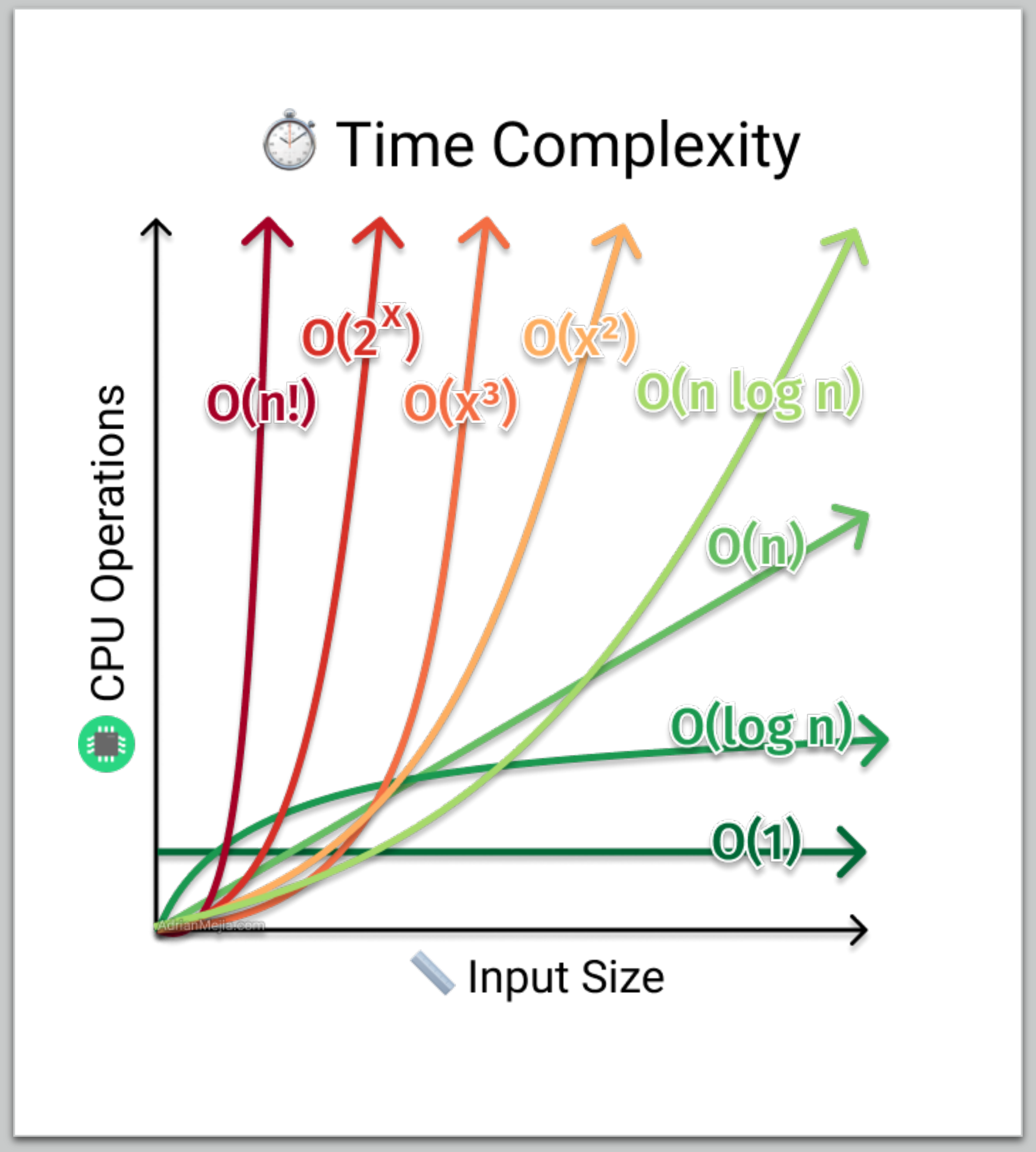
Running time is linear in n, i.e., $T(n) = cn$

SUMMARY

- Analysis of Algorithm: Fibonacci numbers
  - Slow implementation
  - Fast implementation

- Time Complexity
  - How to measure running time?
  - Illustrative examples

- **Asymptotic Notation (O, Ω, Θ)**
  - **Asymptotic Comparison**
  - **Asymptotic Bound**



Asymptotic Notation

- In order to compare running times of algorithms we want a way of comparing the growth rates of functions.

- We want to see what happens for large values of n– small ones are not relevant.

- We are not usually interested in constant factors and only want to consider the dominant term.

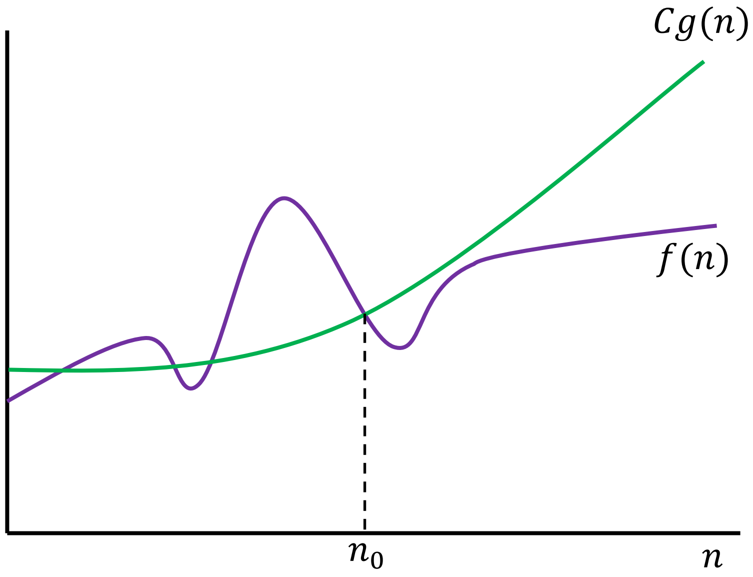
- The standard mathematical approach is to use asymptotic notation $\boldsymbol{O}, \boldsymbol{\Omega}, \boldsymbol{\Theta}$ which we will now describe.

Asymptotic Notation (Contd.)

- Suppose that f and g are functions from $\mathbb{N}$ to $\mathbb{R}$, which take on nonnegative values.

Say f is $O(g)$ ("f is **big-Oh** of g)" if there is some $C > 0$ and some $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $f(n) \leq Cg(n)$. Informally, f grows **at most** as fast as g.

Say f is $\Omega(g)$ ("f is **big-Omega** of g") if g is $O(f)$. Or more formally if there is some $C > 0$ and some $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $f(n) \geq Cg(n)$.
Informally, f grows **at least** as fast as g.



$f(n)$ is $O(g(n))$

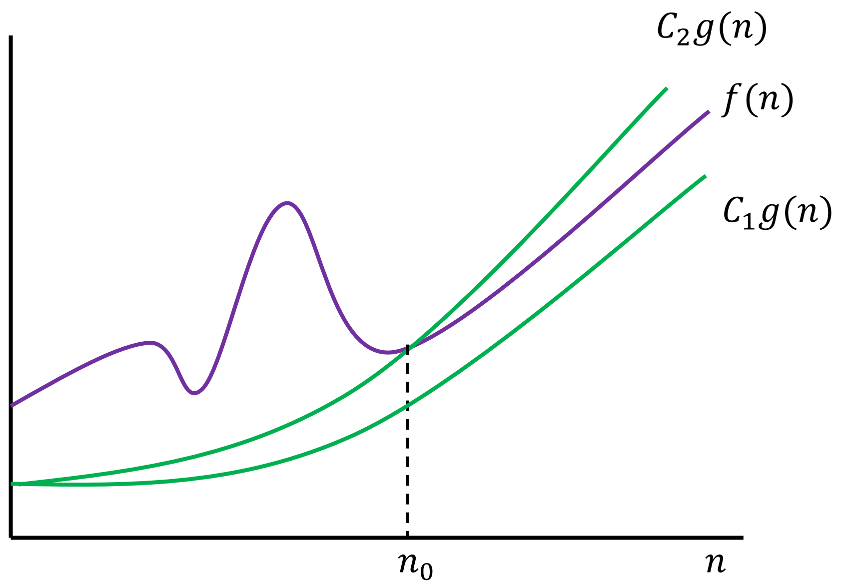
Asymptotic Notation (Contd.)

- Suppose that f and g are functions from $\mathbb{N}$ to $\mathbb{R}$, which take on nonnegative values.

Say f is $\Theta(g)$ ("f is **big-Theta** of g") if f is $O(g)$ and g is $O(f)$. Specifically, if there is some $C_1 > 0, C_2 > 0$, and some $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $C_1 g(n) \leq f(n) \leq C_2 g(n)$.

Informally, f/g is bounded away from zero and infinity, and f grows **at the same rate** as g

Note that we could always reduce n_0 at the expense of a bigger C but it is often easier not to.



$f(n)$ is $\Theta(g(n))$

Asymptotic Comparison

- Every linear function $f(n) = an + b, a > 0$, is $O(n)$.
- **Proof**: $an + b \leq an + |b| \leq (a + |b|)n$ for $n \geq 0$.

- If $f(n) = n$; $g(n) = n^2/2$, then f is $O(g)$ and g is not $O(f)$, so g grows asymptotically faster than f.
- **Proof**: $f(n) \leq 2 \cdot g(n)$ for $n \geq 1$ (or $f(n) \leq 1 \cdot g(n)$ for $n \geq 2$); conversely, suppose that eventually $\frac{1}{2}n^2 \leq Cn$. Then $n \leq 2C$ for all sufficiently large n, a contradiction.

Exercises: Asymptotic Comparison

- I have seen the description "$f = O(g)$ " before. Is this correct?

No. We can't say a function equals to a set of function! $O(g)$ describes a set of functions that grows no faster than g. For instance, both $f_1(n) = n$ and $f_2(n) = 2n$ are in $O(n)$.

But we can say $f \in O(g)$

Rules for Asymptotic Notation

- (**irrelevance of constant factors**) if $c > 0$ is constant then cf is $\mathrm{O}(f)$
- (**sum rule**) If f_1 is $\mathrm{O}(g_1)$ and f_2 is $\mathrm{O}(g_2)$ then $f_1 + f_2$ is $\mathrm{O}(\max\{g_1, g_2\})$
- (**product rule**) If f_1 is $\mathrm{O}(g_1)$ and f_2 is $\mathrm{O}(g_2)$ then $f_1 f_2$ is $\mathrm{O}(g_1 g_2)$
- (**limit rule**) Suppose that $L \coloneqq \lim_{n\to\infty} f(n)/g(n)$ exists. Then,
  - if $L = 0$ then f is $\mathrm{O}(g)$ and f is not $\Omega(g)$
  - if $0 < L < \infty$ then f is $\Theta(g)$
  - if $L = \infty$ then f is $\Omega(g)$ and f is not $\mathrm{O}(g)$.
- (**L'Hopital rule**) is often useful for the application of the limit rule. Note that the limit may not exist at all.

Limit Rule

Suppose that $L := \lim_{n\to\infty} f(n)/g(n)$ exists. Then,

- if $L = 0$ then f is $O(g)$ and f is not $\Omega(g)$;
- if $0 < L < \infty$ then f is $\Theta(g)$;
- if $L = \infty$ then f is $\Omega(g)$ and f is not $O(g)$.

- When f and g are **positive** and **differentiable** functions for $n > 0$, one of the following satisfies:
  - $\lim_{n\to\infty} f(n) = \infty$ and $\lim_{n\to\infty} g(n) = \infty$
  - $\lim_{n\to\infty} f(n) = 0$ and $\lim_{n\to\infty} g(n) = 0$

  L'Hopital rule of calculus can be applied:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

Exercises: Asymptotic Comparison

Prove that exponential functions grow faster than powers:

$$n^k \text{ is } O(a^n) \text{ for all } a > 1 \text{ and } k \geq 0$$

- **Proof**:
- First derivatives of $f(x) = n^k$ by x is $f'(x) = kx^{k-1}$
- First derivatives of $g(x) = a^x$ by x is $g'(x) = a^x \ln a$
- For $k = 0$, $n^k = 1 \in O(a^n)$
- For $k > 0$, by limit rule, and successive application of L'Hopital rule :

$$\lim_{n\to\infty} \frac{n^k}{a^n} = \lim_{n\to\infty} \frac{kn^{k-1}}{a^n \ln a} = \lim_{n\to\infty} \frac{k(k-1)n^{k-2}}{a^n(\ln a)^2}$$

$$\cdots$$

$$= \lim_{n\to\infty} \frac{k!}{a^n(\ln a)^k} = 0 \xrightarrow{\text{When } a > 1} n^k \in O(a^n)$$

Exercises: Asymptotic Comparison

- Logarithmic functions grow slower than powers?

$$\log_a n \text{ is } O(n^k) \text{ for all } a > 1 \text{ and } k \geq 1$$

- **Proof:**
- First derivatives of $f(x) = \log_a x$ when $a > 1$ is $f'(x) = \frac{1}{x \ln a}$
- First derivatives of $g(x) = x^k$ by x is $g'(x) = kx^{k-1}$
- By limit rule and L'Hopital rule :

$$\lim_{n\to\infty} \frac{\log_a n}{n^k} = \lim_{n\to\infty} \frac{\frac{1}{n \ln a}}{kn^{k-1}}$$

$$= \lim_{n\to\infty} \frac{1}{k \ln a \, n^k} = 0 \xrightarrow{\text{When } k > 0} \log_a n \in O(n^k)$$

Time Complexity

- **Informal definition**: A function $f(n)$ such that the running time $T(n)$ of a given algorithm is $\Theta(f(n))$ measures the time complexity of the algorithm.

- An algorithm is called polynomial time algorithm if its runtime $T(n)$ is $O(n^k)$, where k is a constant positive integer.

- A computational problem is called intractable if and only if no deterministic polynomial time algorithm can solve it.

Asymptotic Bounds

- Asymptotic notation measures the running time of the algorithm in terms of elementary operations, i.e., asymptotic bounds are independent of implementation:

- For a given problem, the running time varies NOT ONLY according to the size of the input, BUT ALSO the input itself.
  - E.g., sorting an already sorted array takes almost no time for some sorting algorithms

- Common measures:
  - the worst-case running time
  - the average-case running time

Example: "Find the position of an element x in array"
- Worst Case: x is not in the array
- Average Case: treat all possible positions equally distributed

SUMMARY

- Analysis of Algorithm: Fibonacci numbers
  - Slow implementation
  - Fast implementation

- Time Complexity
  - How to measure running time?
  - Illustrative examples

- Asymptotic Notation $(\mathbf{O}, \mathbf{\Omega}, \mathbf{\Theta})$
  - Asymptotic Comparison
  - Asymptotic Bound